Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

# Implementación en Arquitectura Heterogénea de Sistema de Procesamiento para Cámaras de Eventos

Autor: Rodrigo Gordillo Durán

Tutor/es: José Ramiro Martínez de Dios, Fernando Muñoz Chavero, Aníbal Ollero Baturone

**Dpto. Ingeniería de Sistemas y Automática**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2024

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

# Implementación en Arquitectura Heterogénea de Sistema de Procesamiento para Cámaras de Eventos


Autor:
Rodrigo Gordillo Durán

Tutor/es:
José Ramiro Martínez de Dios
Catedrático
Fernando Muñoz Chavero
Catedrático
Aníbal Ollero Baturone
Catedrático

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla



Sevilla, 2024

Trabajo Fin de Grado: Implementación en Arquitectura Heterogénea de Sistema de Procesamiento para Cámaras de Eventos

Autor: Rodrigo Gordillo Durán
Tutor/es: José Ramiro Martínez de Dios
Fernando Muñoz Chavero
Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

# Agradecimientos

Quería agradecer, en primer lugar, a mi familia, por apoyarme y preocuparse por mí desde que comencé este trabajo hace ya casi un año.

También a mis amigos y compañeros de la carrera, que me han alegrado muchos días.

A mis compañeros de trabajo, especialmente a Saeed, por interesarse por mi trabajo y mi salud.

A Raúl, por ayudarme y contribuir a este trabajo, y por su interés y preocupación.

A Ramiro y Fernando, mis tutores, por también preocuparse y atenderme cuando lo necesitaba.

A Jorge y José María, por ayudarme desinteresadamente cuando me costaba avanzar.

Y a Aníbal, por darme la oportunidad de aprender y formarme en su grupo de investigación.

Gracias a todos.

*Rodrigo Gordillo Durán*
*Escuela Técnica Superior de Ingeniería*

*Sevilla, 2024*

# Resumen

El trabajo que aquí se presenta está asociado a una beca de colaboración ofrecida por el Ministerio de Educación, Formación Profesional y Deportes, que tiene el propósito de iniciar a estudiantes en el mundo de la investigación. En nuestro caso, el objetivo de la beca es el desarrollo de un sistema avanzado de procesamiento de datos provenientes de cámaras de eventos, que pretende integrarse en ornitópteros; robots aéreos de ala batiente en los que, debido a su forma de moverse, resulta de especial interés el uso este de tipo de cámaras.

Para el procesamiento se ha empleado una arquitectura de cómputo heterogénea que integra tanto CPUs (unidades de procesamiento central; o en inglés: *central processing unit*), como FPGAs (matriz de puertas programables en campo; o en inglés: *field-programmable gate array*). Esta combinación permite a nuestra aplicación aprovechar las ventajas de los dos dispositivos: la flexibilidad y facilidad de programación de las CPUs, junto el paralelismo y eficiencia energética de las FPGAs.

Para hacer esto posible, se ha diseñado un marco de trabajo (*framework*) que permite a desarrolladores abstraerse de las complejidades que subyacen a estos sistema electrónicos, de forma que puedan centrarse en la programación de nuevos bloques lógicos de procesamiento sin necesidad de tener conocimientos profundos de todo el sistema. La capa de abstracción que se ha desarrollado resulta especialmente útil en aplicaciones de rápida adaptación y fácil prototipado, como la visión por computador.

Además, para ilustrar el funcionamiento del sistema, se propone una adaptación de un algoritmo para la detección de esquinas a partir de eventos. Estos algoritmos son esenciales para el preprocesamiento de datos, y es que en aplicaciones como la robótica aérea o visión por computador, los tiempos de cómputos son críticos. La adaptación del algoritmo, implementado en la FPGA, permite aprovechar las capacidades que ofrece la lógica programable.

En definitiva, se ha desarrollado un sistema de procesamiento de eventos en una arquitectura heterogénea de computación; de forma que el tratamiento intensivo de datos se hará en la FPGA y las tareas de control y gestión en las CPUs, acelerando así el procesamiento y dando mejores resultados en velocidad y latencia.

# Abstract

The work presented here is associated with a collaboration grant offered by the Ministerio de Educación, Formación Profesional y Deportes, which aims to introduce students to the world of research. In our case, the purpose of the grant is to develop an advanced data processing system using event cameras, intended to be integrated into ornithopters: flapping-wing aerial robots for which, due to their mode of movement, the use of this type of camera is of particular interest.

For processing, a heterogeneous computing architecture has been used, integrating both CPUs (central processing units) and FPGAs (field-programmable gate arrays). This combination allows our application to take advantage of the strengths of both devices: the flexibility and ease of programming of CPUs, along with the parallelism and energy efficiency of FPGAs.

To make this possible, a framework has been designed that allows developers to abstract from the complexities underlying these electronic systems, so they can focus on programming new logical processing blocks without needing deep knowledge of the entire system. The abstraction layer developed is particularly useful in applications requiring rapid adaptation and easy prototyping, such as computer vision.

Additionally, to illustrate the operation of the system, an adaptation of a corner detection algorithm based on events is proposed. These algorithms are essential for data preprocessing, as computation times are critical in applications such as aerial robotics or computer vision. The adaptation of the algorithm, implemented in the FPGA, takes advantage of the capabilities offered by programmable logic.

In summary, an event processing system has been developed within a heterogeneous computing architecture, wherein intensive data processing is performed on the FPGA and control and management tasks are handled by the CPUs, thereby accelerating processing and yielding better results in terms of speed and latency.

# Índice Abreviado

# Índice

# 1 Introducción

Se expondrá a continuación la motivación que hay detrás de este trabajo, así como su contribución. Se comentará también el proyecto en el que se enmarca y la estructura de este documento.

## 1.1 Contexto y motivación

Desde el surgimiento de las FPGA hace tres décadas, su creciente uso se hace patente en diversas ramas de la ingeniería. Estos dispositivos se emplean para implementar circuitos digitales, y ofrecen la solución más económica y rentable a la escalabilidad y al rendimiento.

Existen considerables diferencias entre las FPGA y otros sistemas electrónicos ampliamente empleados como los ASICs (*Application Specific Integrates Circuit*) o los microcontroladores ($\mu$C) y microprocesadores ($\mu$P). Tanto los $\mu$C como los $\mu$P son circuitos integrados programables basados en CPU, lo que los hace notablemente lentos en comparación con las FPGA en ciertas aplicaciones, como el procesamiento de imágenes [2]. Frente al funcionamiento secuencial de las CPU, las FPGA se caracterizan por su estructura concurrente, lo que les permite realizar múltiples operaciones en paralelo. Generalmente, el factor limitante de las FPGAs no es, como en el caso de las CPUs, la velocidad de reloj, sino el número de puertas lógicas disponibles. Los ASICs son circuitos integrados fabricados para un uso particular y que, a diferencia de los otros dispositivos mencionados, no son reprogramables.

Los ASICs ocupan menos superficie que las FPGAs, además de ser más veloces y requerir un menor consumo energético [1] [18]. Sin embargo, las FPGAs son una solución interesante debido a los menores costes de fabricación (especialmente en procesos industriales de pequeña o mediana escala) y tiempo de desarrollo. Debido a la flexibilidad y rápida reconfiguración que ofrecen las FPGA, su uso se ha extendido a múltiples aplicaciones, como lo son la electrónica de consumo, el control industrial, el procesamiento de señales, prototipado rápido o, como es de nuestro especial interés: el procesamiento de imágenes y la robótica.

Uno de los sensores que están emergiendo en el mundo de la robótica es la cámara de eventos. Estas cámaras destacan por su alta velocidad y baja latencia gracias a su funcionamiento asíncrono, tienen un amplio rango dinámico, resistencia al *blur* y bajo consumo energético, dado que solo se transmiten datos cuando se producen eventos, aliviando así la carga computacional. Sin embargo, las cámaras de eventos presentan los inconvenientes de la complejidad del procesamiento, la falta de información en escenas estáticas, el coste, y la escasez de herramientas y soporte para trabajar con ellas; y es que las cámaras de eventos son aún un objeto incipiente en investigación. Las aplicaciones, por otro lado, son diversas, ya que pueden emplearse en vehículos autónomos, visión por

computador, vigilancia, robótica, etc.

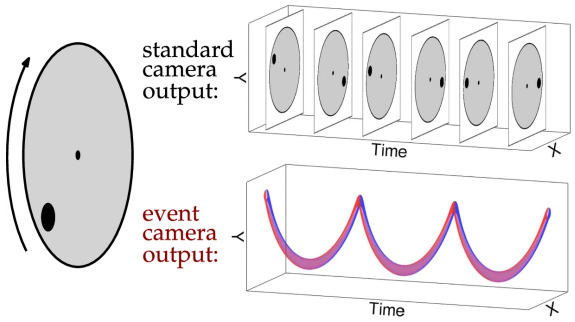



**Figura 1.1** Salida de cámara de eventos vs salida de cámara digital convencional. Figura tomada de Spectra. Accedido el 31 de agosto de 2024.

Las cámaras de eventos se emplean también en ornitópteros, aunque, debido a los movimientos bruscos producidos por el aleteo del robot, se origina una cantidad masiva de eventos que han de ser procesados, pudiendo producirse tasas del orden de decenas de millones de eventos por segundo. Se requiere, pues, un sistema de cómputo capaz de soportar altas velocidades de procesamiento, y que además pueda integrase en el robot. Para poder volar, los ornitópteros deben tener un peso reducido, haciendo así que se recurran a elementos con bajo peso, incluido el sistema embebido de procesamiento. Además, para cumplir las estrictas especificaciones de peso, se emplean fuentes de alimentación de bajo peso; y, por tanto, con poca capacidad, haciendo deseable que el sistema de cómputo sea lo más eficiente posible en términos de consumo energético. Por todo esto, surge la necesidad de recurrir a arquitecturas como las FPGAs.

Y, debido a que las cámaras de eventos suelen emplear una interfaz USB, haremos uso de sistemas electrónicos heterogéneos (*Zynq Ultrascale+ MPSoC*, concretamente), que además nos permitirán explotar las características del procesamiento secuencial y paralelo de forma simultanea.

## 1.2 Contribución

Este trabajo ha resultado en el desarrollo de un *framework* que permite realizar procesamiento de eventos utilizando arquitecturas de computación heterogéneas. Gracias a esta infraestructura, se pueden crear soluciones más eficientes para aplicaciones que emplean cámaras de eventos, donde el tiempo de cómputo es crítico debido al gran ancho de banda de datos que manejan estas cámaras. Este *framework* no solo mejora la eficiencia del procesamiento, sino que también facilita la integración de diferentes tecnologías de *hardware* para optimizar el rendimiento.

Además, se ha programado un entorno que permite emular la cámara, de forma que no sea necesario disponer de una para comprobar y validar el funcionamiento del sistema. Además, para poder ponerlo a prueba, se ha diseñado un bloque de procesamiento implementado en hardware que realiza detección de esquinas.

Finalmente, se han realizado experimentos con la implementación de nuestro método, y se compara el desempeño de éste con otros desarrollados con el mismo objetivo: detección de esquinas en cámaras de eventos.

Se adjuntarán a este documento, en formato digital, todos los códigos y una guía detallada que permiten la reproducción del proyecto. Esta documentación será de gran utilidad para otros desarrolladores e investigadores que trabajen en aplicaciones con características similares, facilitando la adaptación y aplicación del sistema en nuevos proyectos de características similares.

## 1.3 Proyecto en el que se enmarca

Este trabajo se ha realizado dentro del marco del GRIFFIN ERC Advanced Grant (Acción 788247) (griffin-erc-advanced-grant.eu) financiado por el Consejo Europeo de Investigación. Se obtuvo financiación parcial del Proyecto Español ROBMIND (PDC2021-121524-I00) y del Plan Estatal de Investigación Científica y Técnica y de Innovación del Ministerio de Universidades del Gobierno de España (FPU19/04692). El objetivo del proyecto es el desarrollo de ornitópteros. Ver este enlace.

**Figura 1.2** Robot de ala batiente desarrollado en el proyecto GRIFFIN. Imagen tomada de GRIFFIN ERC Advanced Grant. Accedido el 31 de agosto de 2024.

Los robots de ala batiente muestran un gran potencial debido a su seguridad y a su bajo consumo respecto a otros robots aéreos. Los ornitópteros permiten llevar a cabo labores colaborativas con humanos con un menor riesgo que los robots aéreos con propulsores. Además, al no tener hélices, son más resistentes al contacto físico con el entorno, y el ruido que producen es menor, lo que los hace menos invasivos para la naturaleza y deseables para aplicaciones de muestreo o monitorización. En la Fig. 1.2 se muestra un robot de ala batiente desarrollado en el GRVC (Grupo de Robótica, Visión y Control) con un peso de unos 500$g$ y capaz de soportar cargas de su mismo peso.

## 1.4 Estructura del documento

La estructura de este documento pretende exponer de forma clara y ordenada todas las partes que integran el proyecto, y mostrar una visión, primero global, y después específica, del tema que trata.

Comenzaremos con el **Estado del arte**, en el que se recogen los últimos avances producidos en la materia que nos ocupa. También veremos que nuestro trabajo trata dos campos incipientes en investigación: cámaras de eventos y sistemas electrónicos heterogéneos.

Continuaremos con **Descripción del método**, en el que se describen las diferentes capas de abstracción que debemos realizar para llevar a cabo nuestra aplicación de forma eficiente. Veremos además cómo se produce el flujo de datos, desde que se reciben de la cámara de eventos hasta que se visualizan las esquinas en forma de imagen.

En ***Framework*** ahondaremos en los componentes tanto *hardware* (programado) como *software* que hacen posible el procesamiento de los eventos.

El capítulo **Detector de esquinas** expone el algoritmo empleado para el procesamiento de eventos, así como su adaptación para la implementación eficiente en FPGAs.

Para comprobar el correcto funcionamiento del sistema, se verán en el capítulo **Evaluación** las simulaciones y experimentos realizados, y comentaremos brevemente los resultados.

Finalmente, en **Conclusión y trabajos futuros** aporto una reflexión final sobre los resultados y desarrollo del proyecto, así como sobre los posibles trabajos que pueden suceder al que aquí se presenta.

# 2 Estado del arte

En este capítulo llevaremos a cabo una revisión del estado del arte de las tecnologías empleadas en este trabajo, con el objetivo de proporcionar un panorama completo y actualizado de los desarrollos más recientes en este campo.

Comenzaremos explorando las últimas innovaciones en arquitecturas basadas en FPGA y microprocesadores, destacando sus características, capacidades, y aplicaciones más relevantes. Posteriormente, abordaremos las cámaras de eventos, revisando su evolución, principios de funcionamiento, y los avances tecnológicos que han permitido mejoras significativas en la captura y procesamiento de datos, especialmente en FPGAs.

## 2.1 FPGA

Las FPGAs (*Field-Programmable Gate Arrays*) han evolucionado notablemente desde su aparición en 1985, convirtiéndose en componentes clave en la electrónica moderna gracias a su flexibilidad y capacidad de programación. Originalmente diseñadas para ser utilizadas como prototipos y en aplicaciones de baja producción, las FPGAs han ganado popularidad debido a su capacidad para reemplazar a los ASICs (Circuitos Integrados para Aplicaciones Específicas) en una amplia variedad de aplicaciones. Hoy en día, las FPGAs se consideran sistemas en chip (SoC) completamente programables, capaces de adaptarse a un extenso rango de usos, desde la computación de alto rendimiento hasta la inteligencia artificial y la tecnología espacial.

Una de las características más destacadas de las FPGAs es su arquitectura reconfigurable. Esta arquitectura se compone de bloques lógicos configurables (CLBs), interconexiones de enrutamiento y bloques de entrada/salida (I/O). Los CLBs son la base de la lógica programable y se encargan de implementar funciones lógicas. Están formados por elementos lógicos básicos (BLEs), que típicamente incluyen tablas de consulta (LUTs) y flip-flops. Las LUTs permiten la configuración de diversas funciones lógicas, mientras que los flip-flops proporcionan capacidades de almacenamiento y sincronización. Las interconexiones de enrutamiento conectan los CLBs entre sí y con los bloques de I/O, permitiendo la flexibilidad y adaptabilidad que caracterizan a estos sistemas.

Existen dos tipos principales de arquitecturas de enrutamiento, como se muestra en la Fig. 2.2: la de estilo isla y la jerárquica [15,20]. La primera, comúnmente empleada en dispositivos comerciales, organiza los CLBs de forma similar a islas rodeadas por bloques de enrutamiento [15], mientras que la segunda agrupa los CLBs en grupos, optimizando la comunicación interna.

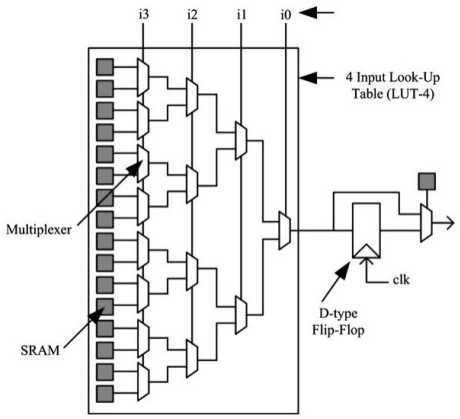
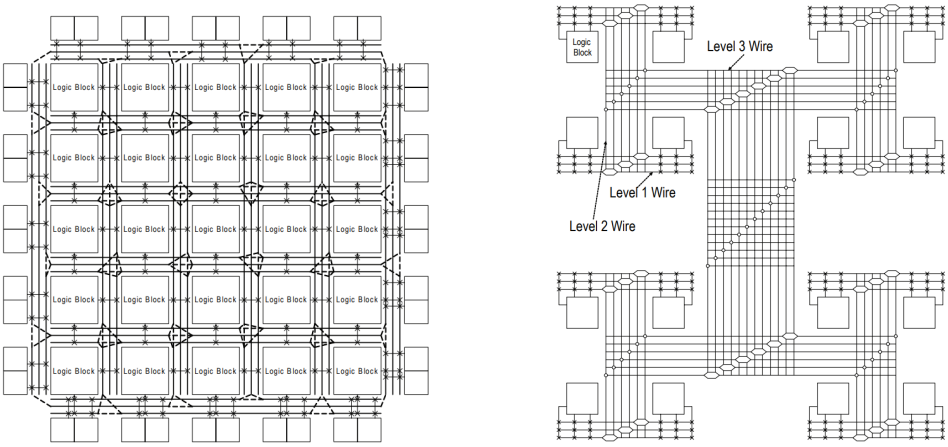



**Figura 2.1** Elemento lógico básico (BLE). Tomada de Electronics Hub. Accedido el 31 de agosto de 2024.




**(a)** Arquitectura estilo isla. **(b)** Arquitectura jerárquica.

**Figura 2.2** Arquitecturas de FPGAs. Figuras tomada de [19].

En cuanto a las tecnologías de programación de las FPGAs, hay varias opciones, cada una con ventajas y desventajas específicas [19]. La tecnología basada en SRAM es la más popular debido a su compatibilidad con la tecnología CMOS y su capacidad de reprogramación. Sin embargo, es volátil, lo que significa que la FPGA debe reprogramarse cada vez que pierde la alimentación. Además, la cantidad de transistores necesarios para cada celda de memoria hace que esta tecnología sea más costosa. La tecnología basada en memoria flash, por otro lado, es no volátil y mantiene la configuración aún cuando se desconecta la alimentación, aunque tiene un límite de reprogramabilidad debido al desgaste del *hardware*, limitando su vida útil y flexibilidad. La tecnología basada en *anti-fuses* utiliza fusibles para almacenar bits, lo que requiere menos área en comparación con las otras tecnologías, sólamente se puede configurar una vez, en la que se queman los fusibles, lo que la hace la menos flexible de las tres tecnologías.

Las aplicaciones de las FPGAs son diversas y abarcan múltiples industrias debido a su capacidad para manejar procesamiento paralelo y tareas intensivas en tiempo real. En el campo de la inteligencia artificial y el aprendizaje automático, las FPGAs son ideales para implementar redes neuronales profundas debido a su capacidad para realizar cálculos de manera paralela y eficiente energéticamente. En la tecnología espacial, las FPGAs se utilizan en satélites y telescopios espaciales, como el Telescopio Espacial James Webb [9], para ajustar interfaces de sensores y

requisitos de procesamiento de imágenes en tiempo real, demostrando su adaptabilidad y capacidad de procesamiento. En sistemas de defensa, las FPGAs se emplean por su capacidad para manejar datos de alta velocidad y su resistencia a manipulaciones [35]. También se utilizan en sistemas de energía renovable, como en la optimización de la orientación de paneles solares y en la detección de gases tóxicos mediante sistemas basados en sensores, contribuyendo a la eficiencia energética y la seguridad medioambiental. Por otro lado, en el experimento Compact Muon Solenoid (CMS), realizado en el Gran Colisionador de Hadrones del CERN, se investigaba el bosón de Higgs y dimensiones adicionales, y para la detección de esta partícula era necesario hacer reconocimiento de patrones, proceso que se implementó en FPGAs.

Desde su introducción, las FPGAs han evolucionado significativamente, expandiendo su cobertura de aplicaciones y mejorando su eficiencia y flexibilidad. Aunque presentan ciertas limitaciones en términos de tamaño y velocidad en comparación con los ASICs, su capacidad de reprogramación y su adaptabilidad las convierten en una herramienta invaluable en diversas industrias, incluyendo la inteligencia artificial, la tecnología espacial y los sistemas de defensa. A medida que la tecnología continúa avanzando, es probable que las FPGAs sigan desempeñando un papel crucial en el desarrollo de nuevas aplicaciones y soluciones tecnológicas, integrándolas junto con otras arquitecturas de computación como los MPSoC.

## 2.2 MPSoC

Los Sistemas en Chip Multiprocesador (MPSoC, por sus siglas en inglés) han experimentado una notable evolución a lo largo de las últimas dos décadas, impulsados por la necesidad de gestionar sistemas cada vez más complejos y con mayores capacidades. A medida que las demandas para el procesamiento avanzado y las aplicaciones de alto rendimiento han aumentado, la gestión de estos sistemas se ha vuelto significativamente más intrincada. Esta complejidad surge de la incorporación de un número creciente de elementos de procesamiento y recursos, lo que conduce a una mayor heterogeneidad en todo el sistema. En este contexto, la gestión eficiente de los MPSoC es fundamental para satisfacer las crecientes demandas de aplicaciones modernas como el Internet de las Cosas [12], la inteligencia artificial y los sistemas digitales basados en la nube [32].

El desafío de gestionar un MPSoC se complica aún más debido a la necesidad de supervisar y controlar múltiples elementos físicos, como procesadores, memorias, puertos y infraestructuras de comunicación, así como factores no físicos, como tareas de proceso, tiempos de utilización, datos y ancho de banda. La adición de más recursos y componentes de comunicación puede aumentar el consumo de energía, las variaciones de temperatura y la vulnerabilidad a diferentes tipos de fallos.

La integración de la gestión inteligente en los MPSoC implica la supervisión y configuración de funcionalidades de control a través de diversos servicios, utilizando técnicas y herramientas que permitan flexibilidad, reconfigurabilidad y adaptabilidad en tiempo de ejecución. Algunos esquemas de gestión propuestos incluyen conceptos novedosos como redes cognitivas [10, 34], sistemas autoconscientes y redes en chip definidas por *software*.

## 2.3 Cámara de eventos

Por otro lado, las cámaras de eventos son una tecnología avanzada inspirada en el sistema de visión biológica [21], diseñado para optimizar la percepción de imágenes y el procesamiento de informa-

ción. A diferencia de los sistemas tradicionales de captura de imágenes que trabajan a una tasa fija de brillo y requieren que todos los píxeles tengan un tiempo de exposición global, las cámaras de eventos funcionan registrando cambios de intensidad de luz en el momento exacto en que ocurren. Esto se logra mediante sensores neuromórficos [13] que imitan los mecanismos de procesamiento visual del cerebro y la retina, permitiendo una detección más eficiente y precisa de cambios en el entorno visual, ver Fig. 2.3.

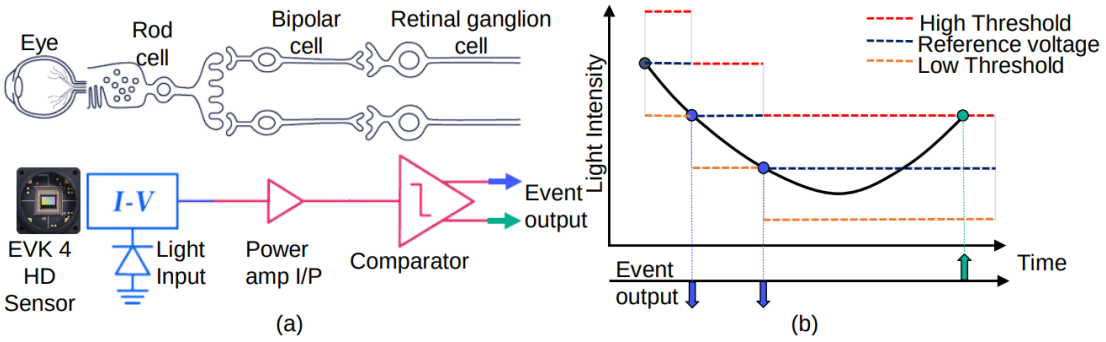



**Figura 2.3** Funcionamiento de cámaras de eventos: (a) Operación independiente de los píxeles. (b) Generación de los eventos. Tomada de [7].

Este enfoque confiere a las cámaras de eventos varias ventajas sobre las cámaras basadas en fotogramas. Debido a que cada píxel en una cámara de eventos se activa y envía datos solo cuando detecta un cambio significativo de brillo (ya sea un aumento o una disminución), estas cámaras son capaces de capturar objetos en movimiento y escenas dinámicas con una resolución temporal extremadamente alta, superando los 10,000 fotogramas por segundo [7]. Como resultado, son especialmente efectivas para minimizar el desenfoque de movimiento (*blur*), un problema común en las cámaras tradicionales [8]. Esta capacidad de respuesta inmediata las hace ideales para aplicaciones que requieren monitoreo en tiempo real y respuestas rápidas, como la robótica [14] o la conducción autónoma [22].

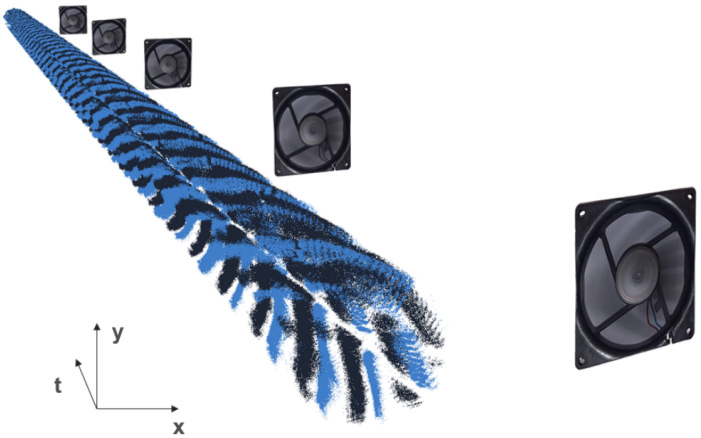



**Figura 2.4** Salida de la cámara que recoge eventos producidos por el movimiento de un ventilador. Tomada de [6].

Además, aunque las cámaras tradicionales pueden alcanzar altas tasas de fotogramas, esto suele

implicar un aumento en los requisitos de ancho de banda y almacenamiento debido al gran volumen de datos que generan. En contraste, las cámaras de eventos solo registran los cambios en la escena, lo que resulta en una cantidad significativamente menor de datos. Esta reducción en el volumen de datos no solo alivia la carga de almacenamiento y ancho de banda, sino que también hace que estas cámaras sean ideales para sistemas embebidos por sus recursos de procesamiento, memoria y almacenamiento limitados. Al centrarse en los cambios en lugar de los niveles absolutos de luz, estas cámaras también reducen la redundancia, capturando solo la información más relevante.

Otra ventaja importante de las cámaras de eventos es su capacidad para operar eficazmente en una amplia gama de condiciones de iluminación [27,37]. Estos sensores son capaces de evitar problemas comunes como la sobreexposición o la subexposición, así como las variaciones repentinas en las condiciones de iluminación que afectan a las cámaras tradicionales. Su amplio rango dinámico, que excede los 120 dB, supera significativamente el rango dinámico de las cámaras basadas en fotogramas de alta calidad, que generalmente no superan los 95 dB. Esta capacidad para manejar entornos con iluminación desafiante, como escenas al aire libre con variaciones de luz, junto con su habilidad para funcionar con niveles de luz extremadamente bajos, ha llevado a exploraciones adicionales en aplicaciones con poca luz.

En el desarrollo de las cámaras de eventos, se distinguen varios tipos de sensores que imitan diferentes aspectos de la visión biológica. El sensor de visión dinámica (DVS) es conocido por emular las funciones de la retina humana simplificada, y basándose en el flujo de información visual a través de diferentes tipos de células neuronales [23]. Otros sensores, como los sensores de imágenes asincrónicos basados en tiempo (ATIS) y los sensores de visión de píxel dinámico y activo (DAVIS), combinan la captura de cambios relativos con la información de exposición absoluta [30] o integran capacidades de captura de escenas estáticas con la detección de eventos [4], respectivamente.

Los sensores de visión neuromórficos, retinas de silicio, o cámaras de eventos, generan flujos de eventos, donde cada evento $e_i$ se describe por:

$$e_i = (x_i, t_i, p_i),$$

donde $x_i$ representa la posición del píxel en coordenadas $(x_i, y_i)$, $t_i$ es la marca de tiempo en la que ocurre el evento, y $p_i$ indica la polaridad del evento, con $p_i \in \{1, -1\}$, que refleja un aumento o disminución de la intensidad.

En conclusión, las cámaras de eventos representan un avance significativo en la captura de imágenes, especialmente en escenarios de alta velocidad o condiciones dinámicas. Su capacidad para reducir la latencia, minimizar el desenfoque de movimiento, disminuir la carga de datos y operar eficazmente en condiciones de iluminación difíciles, las convierte en una opción atractiva para una variedad de aplicaciones, incluyendo la robótica, la vigilancia y los vehículos autónomos, donde el procesamiento rápido y eficiente de la información visual es crucial. Estos dispositivos avanzados, que se asemejan al procesamiento visual biológico, superan las limitaciones de las tecnologías de captura basadas en fotogramas, ofreciendo una mayor eficiencia y precisión en la detección de cambios visuales.

## 2.4 Procesamiento de eventos en FPGA

En los últimos años, las cámaras de eventos han ganado popularidad debido a sus capacidades. La combinación de estas cámaras con las FPGAs ha demostrado ser eficiente en términos de procesamiento en tiempo real y ahorro energético, especialmente en sistemas embebidos. Todos los ejemplos que veremos a continuación se han implementado en sistemas basados en FPGA.

En el campo del procesamiento de eventos en FPGAs, uno de los enfoques importantes es la filtración de datos para reducir el ruido causado por fugas de transistores y fotocorrientes parásitas. Un ejemplo destacado es el trabajo sobre el Background Activity Filter (BAF) [24]: un método que utiliza una matriz de palabras para almacenar marcas de tiempo y comparar eventos entrantes con eventos correlacionados previamente almacenados para distinguir entre eventos reales y ruido. Aunque el filtro BAF ha demostrado ser fundamental para el procesamiento de eventos, su evaluación formal y datos sobre frecuencia y latencia aún son limitados. Otro enfoque relevante es el algoritmo de filtración en cuatro etapas presentado en [5]. Este método aborda el análisis de eventos, detección de coincidencias, agregación-subsampling y codificación de Huffman, utilizando un sensor DVS de 480×320 píxeles. A pesar de su complejidad, el método no ha sido evaluado formalmente, lo que dificulta la comparación de su efectividad con otros métodos de filtración.

Por otro lado, el flujo óptico se refiere al patrón de movimiento aparente de objetos en una escena visual causado por el movimiento relativo entre el observador y la escena. Su análisis permite la detección de objetos en movimiento y la determinación del movimiento propio de la cámara [38]. Uno de los primeros enfoques para implementar flujo óptico en FPGAs es el método basado en *block matching* presentado en [25]. Este enfoque convierte eventos en pseudo-imágenes binarias y utiliza una distancia de Hamming para emparejar bloques de imágenes. A una frecuencia de reloj de 50 MHz, el sistema logró procesar eventos a 5 MEPS (Mega Eventos Por Segundo), aunque la evaluación se realizó sin especificar el conjunto de datos. Otro trabajo importante es el algoritmo de ajuste de planos iterativo para la estimación del flujo óptico [3]. Este método alcanzó una tasa de procesamiento de 2.75 MEPS, suficiente para una resolución de 304×240 píxeles. Para la evaluación, el artículo no ofrece una comparación directa con otros trabajos previos.

La estereovisión es fundamental para la estimación de la profundidad en visión por computadora y presenta una alta complejidad computacional, haciendo interesante su implementación en FPGAs. Un ejemplo destacado es el módulo de estereovisión para detección de caídas, desarrollado en [16]. Este sistema utiliza eventos agregados en pseudo-frames y evalúa la coincidencia usando la diferencia absoluta normalizada en ventanas de 15×15. Aunque la evaluación formal del módulo no está disponible, se realizaron pruebas en un conjunto de datos personalizado para detección de caídas. Otro trabajo [11] relacionado con el anterior mejora el sistema dividiendo la imagen en ventanas de 16×16 y verificando la similitud en disparidad. Este sistema logró una latencia de 13.7 µs por evento, equivalente a 1140 fps.

Otro campo importante en la visión por computador es la detección y clasificación de objetos. Basado en el flujo de eventos, el método PCA-RECT [31] utiliza la extracción de características mediante PCA (Análisis de Componentes Principales). Este método incluye la creación de una matriz de conteo de actividad para cada píxel, del que luego se extraen espacios de dimensiones reducidas para la clasificación mediante un diccionario basado en algoritmos de búsqueda. Este sistema logró una latencia de 550 ns a 100 MHz.

Para el seguimiento de objetos, se ha desarrollado un módulo de seguimiento basado en la detección y seguimiento de clústeres de eventos en [24]. Este módulo se probó en escenarios con objetos

en movimiento lento y rápido. Además, otro trabajo en la misma FPGA [28] propone un modelo de células de ganglio retinal, que detecta objetos en movimiento y puede ser usado para seguimiento, logrando latencias de 220 a 440 ns a 50 MHz.

La investigación sobre la implementación en *hardware* de procesamiento de eventos se encuentra algo rezagada respecto a los desarrollos más recientes en algoritmos de procesamiento de eventos. A partir del análisis realizado en [17], se extrajeron las siguientes conclusiones:

Conectar una cámara DVS e a la lógica reconfigurable de una FPGA no es sencillo, y es que muchas investigaciones han utilizado soluciones personalizadas difíciles o imposibles de replicar. Además, la mayoría de los trabajos procesan datos de DVS con resoluciones muy bajas, y solo unos pocos estudios consideran sensores HD (1280×720). Hasta ahora, la mayoría de los trabajos utilizan imágenes de eventos en lugar de un procesamiento "directo", y existe una falta notable de evaluación en los métodos implementados, sin comparaciones con trabajos anteriores ni basadas en bases de datos ampliamente utilizadas.

Por último, es necesario investigar la implementación de métodos modernos de inteligencia artificial, incluidos los GNNs (redes neuronales basadas en grafos) y *transformers*, y explorar más a fondo los SNNs (redes neuronales espiking) para su uso en dispositivos neuromórficos. Finalmente, también se debería prestar mayor atención desarrollo de sistemas integrales en robótica, como para vehículos autónomos o drones.

# 3 Descripción del método

En este capítulo vamos a introducir la arquitectura *hardware* que hemos empleado, así como las diferentes capas que integran el proyecto. Además, se describirá cómo se produce el flujo de información, desde que se reciben datos hasta que se visualizan los resultados.

## 3.1 Arquitectura Zynq Ultrascale+ MPSoC

La familia de dispositivos *Zynq Ultrascale+ MPSoC* integra una arquitectura constituida por *hardware* programable y múltiples procesadores ARM de 64 bits, permitiendo así un procesamiento heterogéneo que mejora el desempeño y seguridad en aplicaciones capaces de aprovechar las características de las FPGAs y los procesadores ARM. Estos dispositivos pueden incluir, a su vez, unidades para el procesamiento en tiempo real, módulos optimizados para la implementación de redes neuronales convolucionales, módulos duros o blandos para procesamiento de vídeo o de señales, etc.

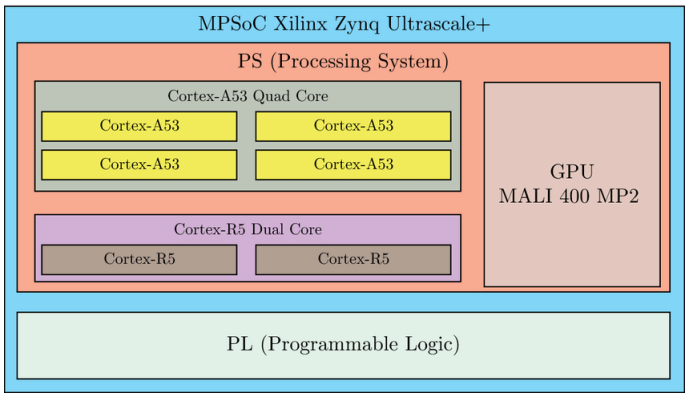



**Figura 3.1** PS y PL en Zynq Ultrascale+ MPSoC. Figura tomada de ResearchGate. Accedido el 31 de agosto de 2024.

En estas arquitecturas distinguimos dos partes: PS y PL, como se muestra en la Fig. 3.1. En el PS (*processing system*) se encuentran los procesadores ARM, núcleos gráficos, unidades de procesamiento en tiempo real, sistemas de seguridad, unidad de manejo de plataforma, memoria e interfaces (depende de la placa de desarrollo, pero normalmente USB, HDMI, SPI, UART, ETHERNET, etc). Por otro lado, el PL (*prorgammable logic*) integra todo el tejido lógico, así como módulos du-

ros o blandos, que dependen de la familia (CG, EG o EV), e interfaces, como se muestra en la Fig. 3.2.

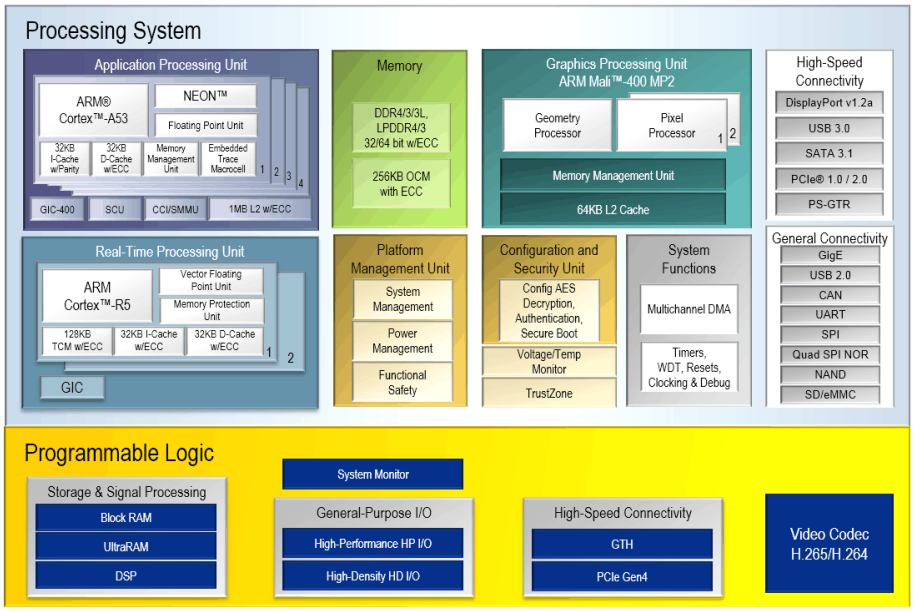



**Figura 3.2** Arquitectura Zynq Ultrascale+, familia EV. Figura tomada de Electronics. Accedido el 31 de agosto de 2024.

Existen también interfaces entre ambas partes, PL y PS, que posibilitan el flujo de datos entre ellas, como se puede ver en la Fig. 3.3, que resulta más descriptiva de esta arquitectura.

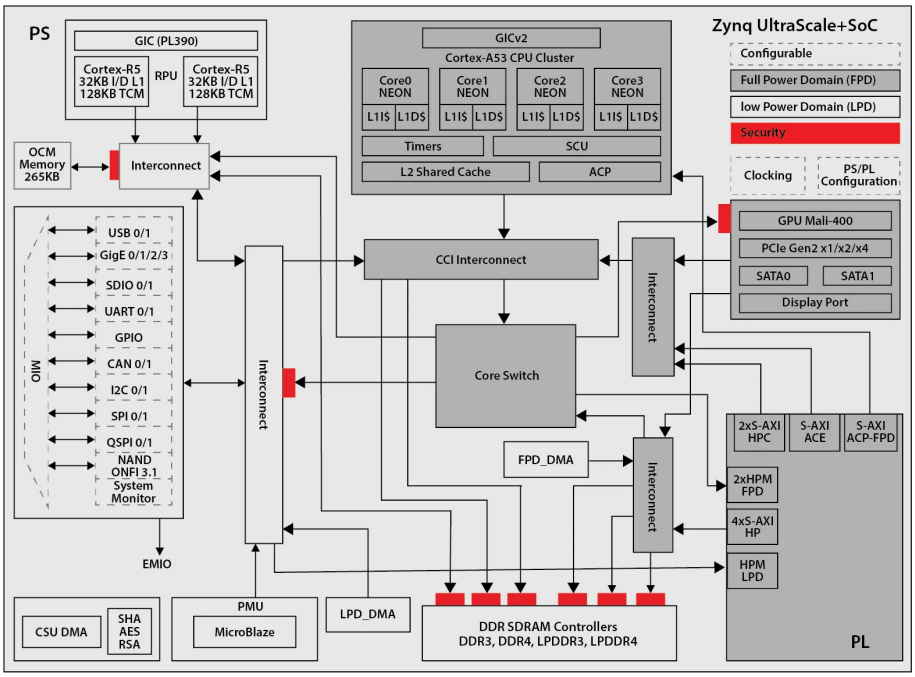



**Figura 3.3** Arquitectura Zynq Ultrascale+. Figura tomada de Packtpub. Accedido el 31 de agosto de 2024.

## 3.2 Flujo de datos

La secuencia que siguen los datos es la que se muestra en la Fig. 3.5.


**Figura 3.4** Flujo de datos.

Desde una cámara de eventos se enviarán paquetes que contienen datos de eventos de 64 bits, codificados como se muestra en la Fig. 3.5, incluyendo la posición en coordenadas $x$ e $y$, la polaridad y la marca de tiempo del evento. Estos datos los recibimos a través de USB directamente en el PS, aunque el procesamiento, por motivos de eficiencia, se hará en el tejido lógico. Para poder enviar datos desde el PS al PL existen diversos métodos que emplean diferentes interfaces, aunque en este trabajo se ha empleado la forma que, hasta donde conocemos, resulta más eficiente para nuestra aplicación: DMA (*Direct Memory Access*).

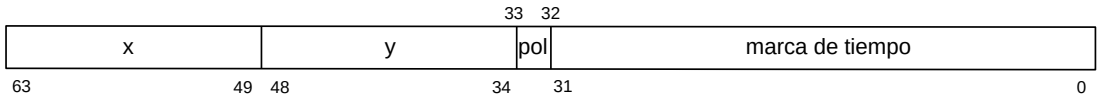



**Figura 3.5** Codificación de un dato de evento.

A través de la DMA seremos capaces de enviar los datos que recibimos de la cámara a la zona lógico-programable de nuestro sistema. Una vez ahí, antes de introducir datos en el bloque de procesamiento, emplearemos una memoria FIFO que reduce la pérdida de datos producida por el tiempo que se tarda en procesar cada dato. Cuando se detecte una esquina, se introducirá en otra memoria FIFO el dato asociado a ésta, que también será de la forma que se muestra en la Fig. 3.5. De esta segunda FIFO se extraerán los datos desde el PS a través de DMA cuando el sistema de procesamiento no esté ejecutando otras tareas más prioritarias. Con los datos de las esquinas podremos realizar más procesamiento o evaluar el correcto funcionamiento del sistema.

La DMA es un componente de los sistemas *hardware* que permite acceder a la memoria sin la intervención de la Unidad Central de Procesamiento (o CPU). De esta forma se evita que la CPU tenga que atender a interrupciones con cada transferencia que se requiera y permitiendo que realice otras tareas, como podría ser, en nuestra aplicación, el filtrado de eventos [33], o procesado posterior a la detección de esquinas que sea más conveniente implementar con algoritmos secuenciales.

## 3.3 Jerarquía *software*

La jerarquía que sigue el proyecto es la que se muestra en la Fig. 3.7. Está compuesta, en primera instancia, por la cámara, que será la que proporcione los datos de eventos a través de USB. El USB estará conectado a una placa de desarrollo, en la que haremos todo el procesamiento de los datos, y, una vez obtenidos los resultados, se enviarán a otro ordenador, que permitirá visualizar las imágenes

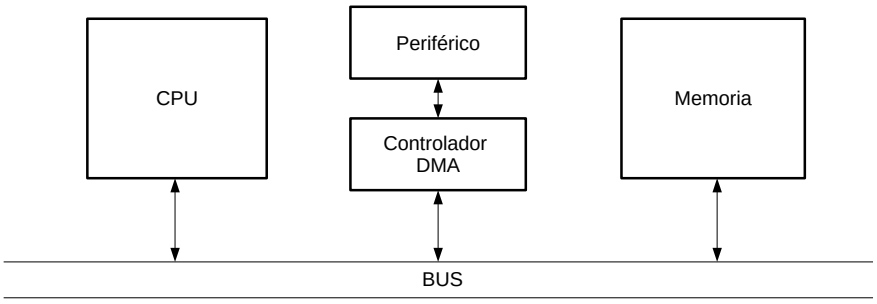



**Figura 3.6** Funcionamiento DMA.

generadas por todos los eventos y, de forma separada, por las esquinas detectadas.

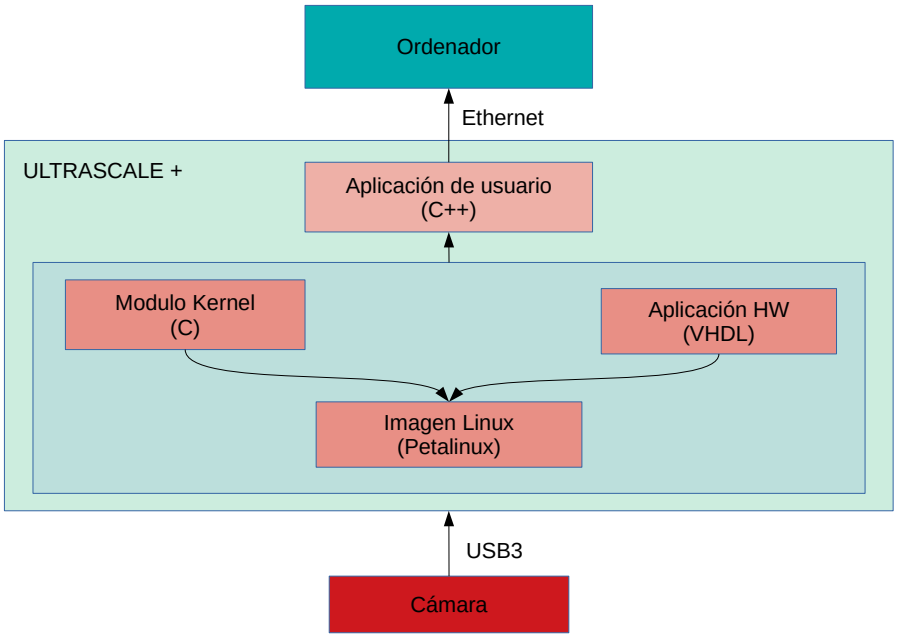



**Figura 3.7** Flujo de datos.

### 3.3.1 Sistema operativo

Dado que la cámara tiene una interfaz USB, necesitamos un *driver* para poder comunicarnos a través de este protocolo. Una forma de abordar la comunicación es implementar el *driver* directamente en una FPGA, aunque debido al funcionamiento inherentemente secuencial del protocolo USB, resulta conveniente aprovechar los *drivers* de un sistema operativo como *Linux*; y es por ello que se ha empleado *Petalinux*.

*Petalinux* es una SDK (*Software Development Kit*) que se emplea para generar distribuciones de *Linux* para sistemas embebidos. Esta herramienta, basada en el *Yocto Project*, está especialmente diseñada para funcionar con los sistemas en chip (SoC) de Xilinx, que, además de FPGAs, integran procesadores ARM o microcontroladores. Esta plataforma permite desarrollar versiones de *Linux* adaptadas a la aplicación, de forma que se ajuste a los requerimientos *hardware* y *software* para una solución más eficiente en términos de consumo energético y de recursos, que son limitados y costosos en estas arquitecturas.

A las funcionalidades básicas que incluye *Petalinux* para generar la imagen de sistema operativo se han tenido que añadir otras que nos permitan poner en marcha nuestra aplicación, como *FPGA Manager* (se menciona su uso en la guía que acompaña a este documento), *opencv*, compiladores, X11, o ficheros de cabecera para compilar módulos kernel, entre otras.

### 3.3.2 Módulo kernel

Un módulo kernel es un fragmento de código que se puede cargar y eliminar dinámicamente en el sistema operativo a nivel del kernel, y que permite, entre otras cosas, manejar interrupciones producidas por componentes *hardware*, tarea que sólo puede gestionarse eficientemente de esta forma.

En nuestro proyecto se ha programado un módulo que sirve como *driver* de las comunicaciones entre el PL y PS a través de la DMA, como comentaremos en el Capítulo 4. Esto es necesario ya que el bloque de DMA genera interrupciones al finalizar las transferencias, y cuando éstas se produzcan debemos realizar nuevas escrituras o lecturas de datos.

### 3.3.3 Aplicación *hardware*

Por otro lado, debemos programar la aplicación *hardware* que incluirá, entre otros bloques, el de procesamiento de los eventos. Esta aplicación será la que se ejecute en el tejido lógico, y con la que programaremos la FPGA.

La aplicación creará componentes que desde el punto de vista del sistema operativo se comportan como periféricos, por lo que deben ser incluidos en el *device tree*, indicando, entre otras cosas, el *driver* que los gestionará. De esta forma, al bloque de DMA, implementado en la FPGA, le asociamos el *driver* que hemos creado con el módulo kernel mencionado anteriormente.

### 3.3.4 Aplicación de usuario

La aplicación de usuario permitirá hacer la configuración inicial de las comunicaciones a través de la DMA, así como recibir los datos de los eventos desde la cámara y de las esquinas desde la FPGA. Además, para poder depurar la aplicación así como observar los resultados, se enviarán tanto los datos de los eventos como los de las esquinas a otro ordenador a través de *sockets*. Hemos empleado otro ordenador, y no la propia placa de desarrollo, para evitar instalar gráficos en la imagen del sistema operativo, que lo haría más lento y aumentaría su consumo.

En la versión final de este proyecto no se espera tener HDMI en la placa de desarrollo, por lo que simplemente sería necesario excluir la parte del programa que manda los datos para que fuera funcional, y sin haber tenido que incluir gráficos para depurar.

### 3.3.5 Visualizador

En ese otro ordenador crearemos dos clientes que reciban los datos a través de los *sockets*, y cuando alcancemos una cantidad determinada de esquinas, mostraremos la imagen con la cantidad de éstas indicada; y, en otra ventana, los eventos recibidos desde que se mostró la anterior imagen.

# 4 Framework

En este capítulo vamos a señalar cuáles son los componentes necesarios para que se produzca de forma eficiente y fiable el flujo de datos desde que se reciben a través del USB desde la cámara hasta que se obtienen los datos procesados por el PL de nuevo en el PS.

## 4.1 Aplicaciones de usuario

En la Fig. 4.1 se muestra la interacción entre los diferentes hilos de las aplicaciones de usuario, tanto en la placa de desarrollo como en el ordenador externo.

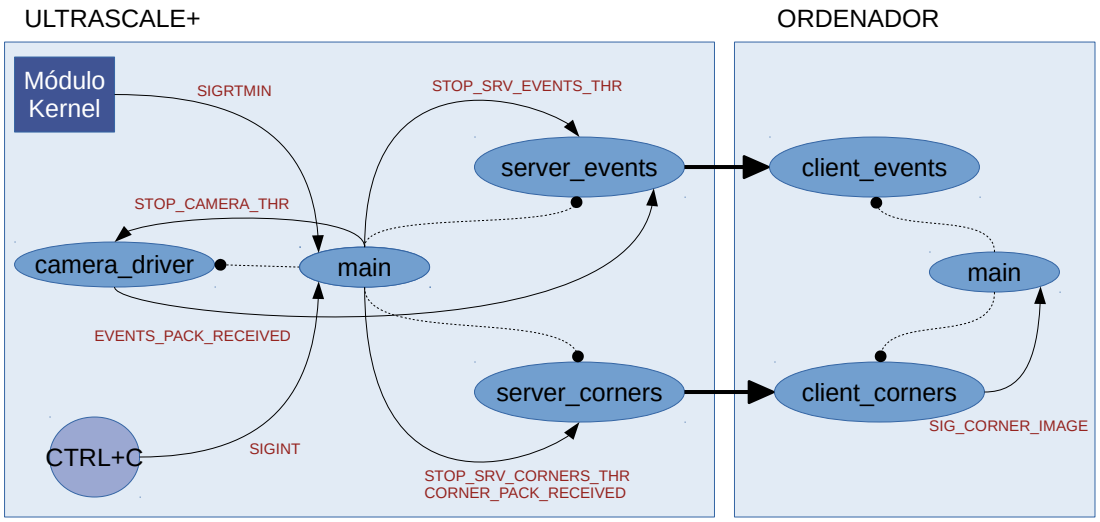



**Figura 4.1** Comunicación entre hilos y procesos de las dos aplicaciones de usuario. Las flechas indican el envío de señales.

En la apliación de usuario de la placa de desarrollo hemos empleado 4 hilos: el hilo *camera_driver*, que gestiona la comunicación de la cámara empleando la librería *openev*, desarrollada por Raúl Tapia, para recibir datos de eventos; *server_events*, que manda los datos de los eventos al ordenador externo cuando se reciben desde la cámara; *server_corners*, que envía los datos de las esquinas que se han detectado en la FPGA; y, por último, el principal: *main*, que inicia las comunicaciones, crea los hilos, y los cierra adecuadamente al presionar *CTRL+C* en el terminal.

Además de lo anterior, el hilo principal es el que recibe la señal desde el módulo kernel que indica que un nuevo paquete de esquinas se ha recibido desde la FPGA, y entonces manda la señal correspondiente al hilo *server_corners* para que envíe los datos a través del *socket*. Del mismo modo, el hilo *camera_driver* está en una espera bloqueante hasta que se recibe un nuevo paquete de eventos, y entonces manda una señal al hilo *server_events*, que los reenvía.

Por otro lado, el hilo *main* de la aplicación en el ordenador crea otros dos hilos que reciben los datos de eventos y esquinas. Cuando se ha recibido un número determinado de esquinas, se muestran dos imágenes: una con las esquinas y otra con los eventos que se han recibido desde que se recibió el anterior paquete de datos de esquinas.

## 4.2 Aplicación *hardware*

### 4.2.1 Protocolo AXI

Además de los bloques que comentaremos a continuación, se ha empleado el protocolo AXI, un componente fundamental del sistema. AXI (Advanced eXtensible Interface) es un estándar desarrollado por ARM que se emplea como interfaz de comunicación entre componentes dentro de un sistema SoC, haciendo que la comunicación entre ellos sea rápida y eficiente. Este protocolo permite la comunicación entre módulos IPs ofrecidos por compañías (de pago o no) y los diseñados por los desarrolladores, favoreciendo así su portabilidad.

Entre los protocolos AXI se diferencian tres: *AXI Full*, *AXI Lite* y *AXI Stream*, todos recogidos en AMBA, y empleándose cada uno de ellos para determinados tipos de transferencias según convenga en la aplicación. En nuestro trabajo han sido empleados (programados) *AXI Lite* y *AXI Stream*, que comentaremos brevemente.

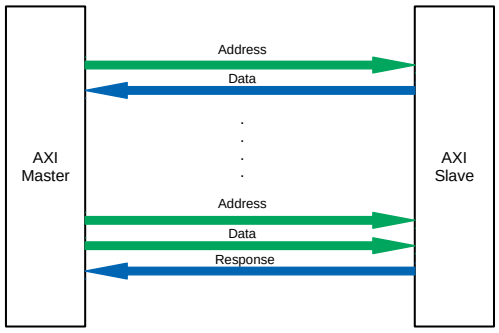



**Figura 4.2** AXI Lite.

*AXI Lite* la emplearemos para poder acceder a cualquier dirección de memoria DDR desde el PS. Las herramientas que permiten programar el dispositivo son capaces de asociar regiones de memoria DDR a registros de configuración de los diferentes módulos, de modo que desde una aplicación ejecutada en el sistema operativo que corre en el PS se puede acceder a registros que forman parte del tejido lógico. Este protocolo sirve para transacciones de lectura o escritura de un solo dato, a diferencia de *AXI Stream*. En la Fig. 4.2 se muestran varias señales de este protocolo.

Por otro lado, *AXI Stream* permite que, cada vez que se realiza una transacción empleando un canal que use este protocolo, una ráfaga de datos sean transferidos en escritura o lectura. Este

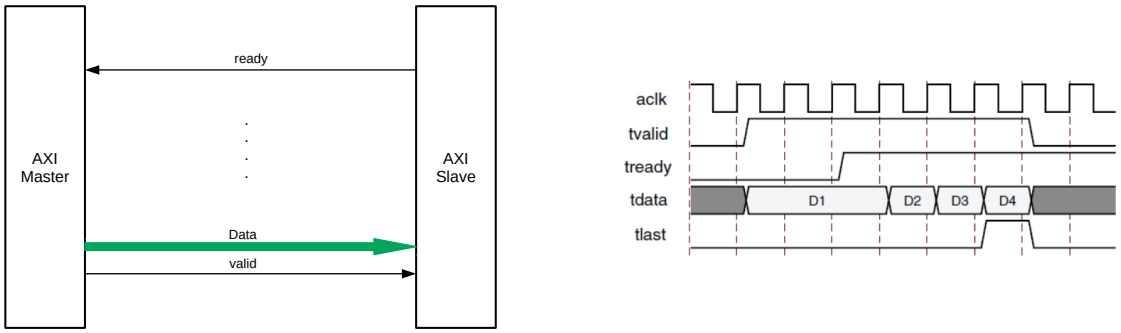



**Figura 4.3** AXI Stream. La figura de la derecha ha sido tomada de AMD. Accedido el 31 de agosto de 2024.

protocolo nos será útil para la transmisión de los datos de los eventos, que son recibidos en forma de paquete a través de USB. De esta forma, cada vez que se reciba un nuevo paquete de eventos, estos se mandarán en serie al tejido lógico para su procesamiento. En la Fig. 4.3 se muestra una sencilla transmisión de datos empleando este protocolo, con las variaciones de algunas de las señales implicadas.

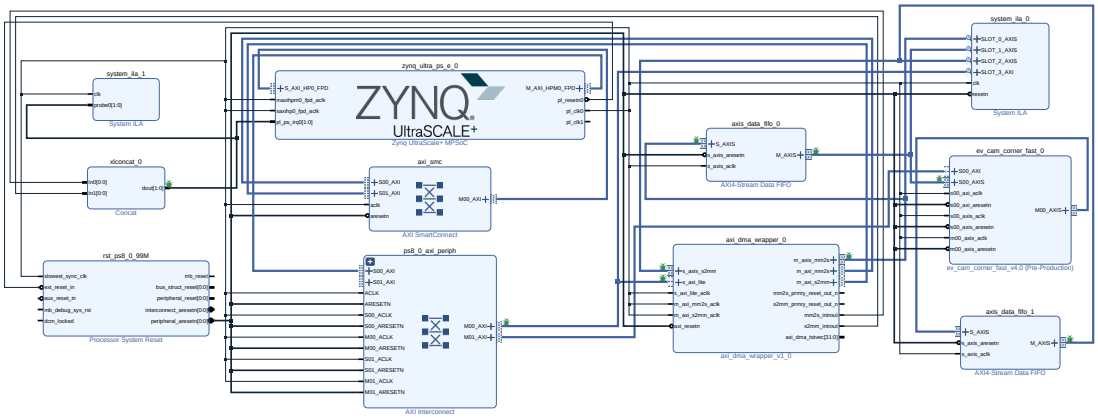



**Figura 4.4** Diagrama de bloques generado en *Vivado*.

### 4.2.2 *Diagrama de bloques*

En este apartado haremos una breve descripción de los bloques que componen el sistema y que se muestran en la Fig. 4.4. Todos ellos, salvo *axi_dma_wrapper* y *ev_cam_corner_fast*, son facilitados por *Xilinx*; no han tenido que ser programados, sólo configurados.

#### *Zynq Ultrascale+ MPSoC*

Este bloque permite instanciar al PS dentro de nuestro sistema. El módulo, básico y necesario en cualquier aplicación que emplee la arquitectura *Zynq Ultrascale+*, sirve como una interfaz que permite conectar el sistema de procesamiento con el tejido lógico diseñado para la aplicación, facilitando así la integración de otros bloques digitales. Este bloque permite, además, configurar las interfaces con periféricos externos (USB, UART, GPIO, etc), los relojes y *resets*, y la memoria, entre otras. (Ver Fig. 4.5).

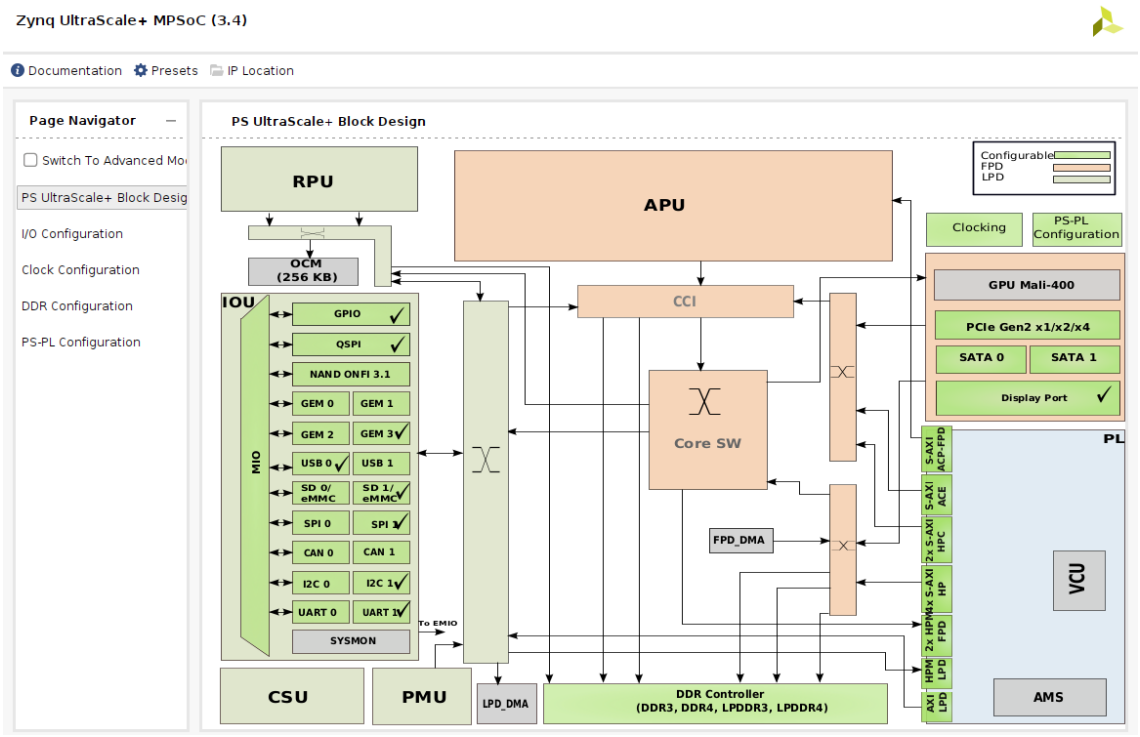



**Figura 4.5** Configuración de *Zynq Ultrascale+ MPSoC* en *Vivado*.

### *AXI Interconnect*

La función de *AXI Interconnect* es la de multiplexar los canales que empleen el protocolo AXI, tanto esclavos como maestros. De esta forma, un mismo maestro puede tener varios esclavos. o un solo esclavo varios maestros.

En nuestra aplicación, el bloque *Zynq Ultrascale+ MPSoC* tiene un canal AXI maestro que configura tanto al bloque de procesamiento (*ev_cam_corner_fast*) como al de DMA (*axi_dma_-wrapper*).

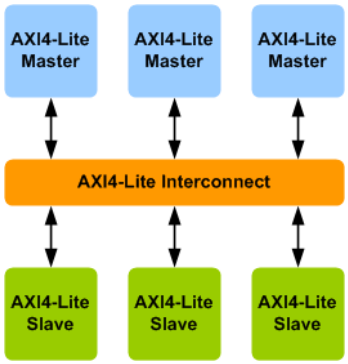



**Figura 4.6** AXI Interconnect. Figura tomada de SmartDV. Accedido el 31 de agosto de 2024.

### *AXI SmartConnect*

El bloque *AXI SmartConnect* tiene el mismo propósito que el anterior. Es una versión más reciente. Lo empleamos para que el puerto esclavo AXI de *Zynq Ultrascale+ MPSoC* tenga dos maestros, ambos canales del bloque *axi_dma_wrapper*.

#### *Processor System Reset*

Genera las señales de *reset* para todos los bloques que componen el sistema de procesamiento.

El módulo permite configurar estas señales, pero en nuestra aplicación se han empleado las que se establecen por defecto.

#### *AXI4-Stream Data FIFO*

Implementa memorias FIFO (*First In Firt Out*), que amortiguan el tiempo de procesamiento de datos a traves del bloque *ev_cam_corner_fast* y también el de transmisión de datos desde o hacia el PS.

#### *System ILA*

Este bloque *System Integrated Logic Analyzer* (*System ILA*) sirve para poder depurar el funcionamiento del sistema, monitorizando señales y registros del tejido lógico. La herramienta de depuración representará el valor de las señales que se quieran evaluar cuando se detecte que alguna de ellas alcance un valor que se haya configurado; por ejemplo, permite observar transiciones en el flujo de procesamiento tras detectar que entra un dato en él. Este bloque, aunque sea para depuración, consume recursos de la FPGA, incluyendo memoria.

#### *axi_dma_wrapper*

Este bloque sirve para envolver al bloque *AXI Direct Memory Access*, también proporcionado por *Xilinx*. Nuestro bloque únicamente une las entradas y la salidas de la IP a las correspondientes del módulo que hemos programado, permitiendo que el sistema operativo creado con *Petalinux* no asocie el driver por defecto al módulo DMA, para así poder emplear el módulo kernel desarrollado por nosotros, específico de nuestra aplicación.

Como se puede apreciar en la Fig. 4.7, el bloque tiene un canal *AXI Lite* de entrada, que permite el acceso a los registros de configuración y a los registros de estado, entre otros. Otros canales importantes son el *AXI4- Stream Master (MM2S)* y el *AXI4- Stream Slave (S2MM)*: el primero convierte los datos almacenados en memoria a una ráfaga de datos (*memory map* to *stream*), que será la que llegue al bloque *ev_cam_corner_fast* a través de una FIFO; y el segnudo recibe una serie de datos que copia en memoria DDR (*stream* to *memory map*).

Desde el módulo kernel que hemos empleado debemos modificar los registros cada vez que queramos realizar una transmisión de datos. Al enviar datos desde el PS al PL emplearemos el canal MM2S, que configuraremos estableciendo en los registros correspondientes la dirección de memoria a partir de la cual queremos mandar los datos, así como la cantidad de estos que habrá en la transmisión. De la misma forma, al emplear el canal S2MM para recibir datos del bloque de procesamiento, debemos indicar la posición de memoria DDR en la que queremos escribir, así como la cantidad de datos que recibiremos.

Además, como se puede apreciar en la Fig. 4.8, existen dos canales, *mm2s_introut* y *s2mm_introut*, que producen interrupciones cada vez se finaliza una transmisión de lectura o escritura de datos en el PL, respectivamente. Estas interrupciones son las que se manejan desde el módulo kernel, que permite gestionar el envío de datos de eventos y la lectura de datos de esquinas desde el PS.

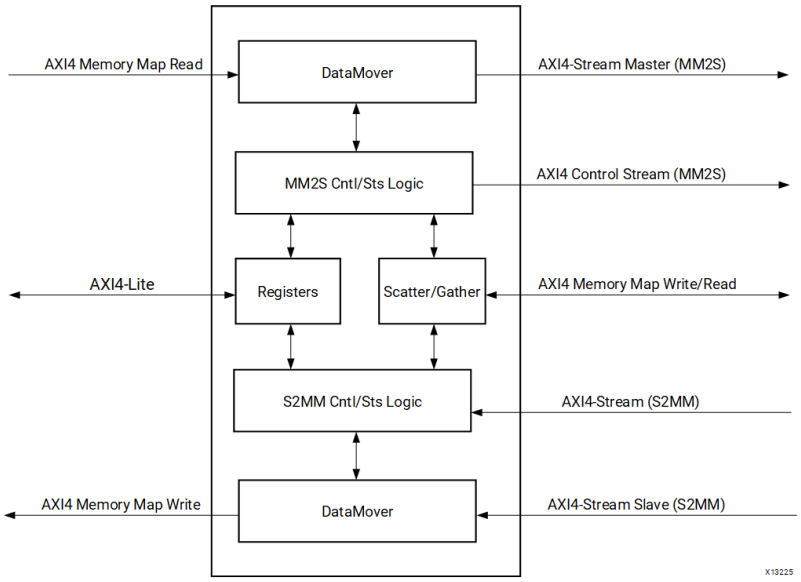



**Figura 4.7** Controlador AXI DMA. Imagen tomada de Lauri's Blog. Accedido el 31 de agosto de 2024.


**Figura 4.8** Bloque *axi_dma_wrapper* generado en *Vivado*.

### *ev_cam_corner_fast*

Este es el bloque que, recibiendo un *stream* de datos de eventos, envia un nuevo *stream* con los eventos que se han identificado como esquinas. La cantidad de esquinas que han de ser detectadas antes de mandar el paquete puede ser configurada junto con otros parámetros (ver Fig. 4.9) que se verán en el Capítulo 5. En todo caso, la cantidad de datos que se transmiten en cada ráfaga no puede exceder los 256, para no acaparar el bus de datos, en el que otros bloques pueden requerir leer o escribir.

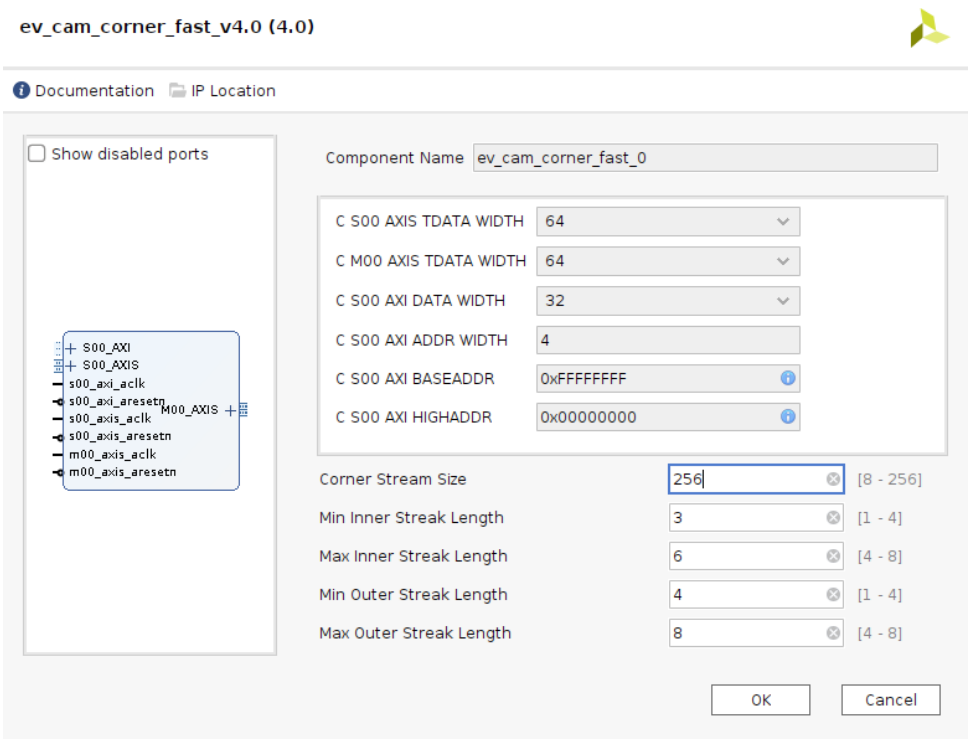



**Figura 4.9** Configuración del módulo de procesamiento en *Vivado*.

# 5 Detector de esquinas

En este capítulo se describirá el algoritmo escogido para la detección de esquinas así como su adaptación para posibilitar una implementación más eficiente en la FPGA, de forma que se aprovechen las características descritas anteriormente de estas arquitecturas.

## 5.1 Algoritmo FAST

El algoritmo FAST (*Features from Accelerated Segment Test*) es ampliamente conocido por su eficiencia en el preprocesamiento de imágenes, ya que evita el uso de derivadas, que son altamente costosas computacionalmente y pueden amplificar el ruido.

Inspirado en el algoritmo FAST, se desarrolló una adaptación para el procesamiento de eventos [29] que requiere, al igual que su versión original, únicamente de comparaciones, lo que lo hace especialmente interesante para su uso en FPGAs.

### 5.1.1 FAST en procesamiento de *frames*

El algoritmo FAST es un método de procesamiento empleado para la detección rápida de características de interés (normalmente esquinas) en una imagen. Para la detección de esquinas, se evalúa la intensidad luminosa de los píxeles que rodean a un píxel centran siguiendo un patrón circular (círculo de Bresenham) de un radio $r$ determinado. El píxel central $p$, con intensidad luminosa $I_p$, se considera una esquina si un conjunto $S$ de $N$ píxeles contiguos tienen una intensidad luminosa mayor que la del píxel central más un umbral $t_1$, o menor que ésta menos un umbral $t_2$, ver Fig. 5.1.

- $\forall x \in S,\ I_x > I_p + t_1$ (esquina oscura)
- $\forall x \in S,\ I_x < I_p - t_2$ (esquina clara)

### 5.1.2 FAST en procesamiento de *eventos*

Una ventaja de las cámaras de eventos frente a las convencionales es que en éstas es necesario procesar todos los píxeles del nuevo *frame*, mientras que los eventos pueden ser procesados individualmente y de forma asíncrona cuando se reciben. El procesamiento de eventos se realiza sobre la marca de tiempo y la polaridad del evento, y no sobre la intensidad luminosa del píxel, por lo que es necesario adaptar el algoritmo antes expuesto.

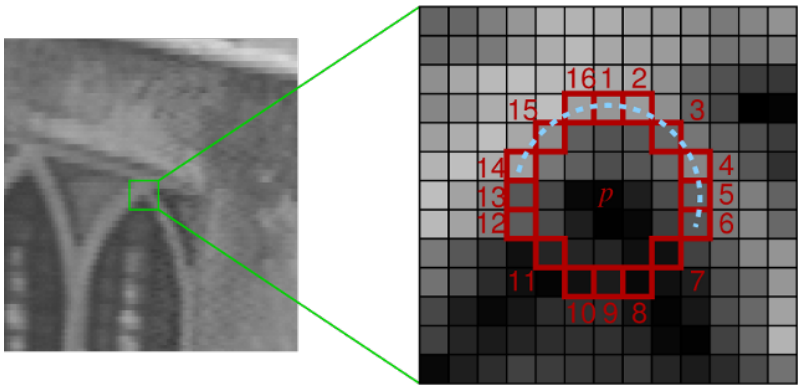
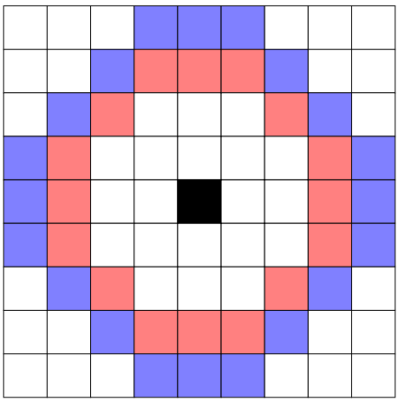


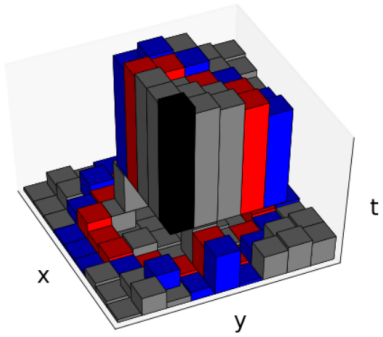

**Figura 5.1** Detector de esquinas FAST en imágenes basadas en *frames*, con $r = 3$ y $n = 9$. Figura tomada de Edward Rosten. Accedido el 31 de agosto de 2024.

En el método descrito en [29] se considera que la marca de tiempo del evento que se va a procesar es siempre mayor que la de los píxeles que lo rodean, por lo que las comparaciones no se realizarán sobre el píxel central, sino entre el resto de píxeles de los círculos de Bresenham. Como en [36], el procesamiento se hace de forma independiente para cada polaridad, habiendo así dos SAEs (Surface of Active Events), cada una asociada a una polaridad.

Por motivos de eficiencia se comprobó experimentalmente que conviene emplear dos circulos de radio $r_1 = 3$ y $r_2 = 4$. El algoritmo busca, en cada círculo, una cadena que contenga $N$ valores consecutivos que sean todos ellos mayores que los del resto del círculo. Para el círculo interior, de radio $r_1 = 3$, $N$ puede tomar valores entre 3 y 6, mientras que para el círculo exterior $N$ vale entre 4 y 8. Emplear más círculos no solo aumentaría el coste computacional, sino que empeoraría la eficiencia del método.



**(a)** Círculos de radio $r_1 = 3$ y $r_2 = 4$.




**(b)** Representación de la superficie de eventos.

**Figura 5.2** Detector de esquinas FAST adaptado a eventos. Figura tomada de [29].

## 5.2 Implementación

Antes de explicar cómo se ha implementado el algoritmo antes expuesto en la FPGA, cabe describir cómo sería para una arquitectura basada en procesadores.

### 5.2.1 Implementación en procesadores

En [29] se hace lo siguiente:

**Algoritmo 1:** Pseudocódigo implementación secuencial

```
Input: evento
Output: es_esquina
Función detecta_cadena:
    actualiza la matriz con marcas temporales
    inicializa es_esquina ← falso
    inicializa no_es_esquina ← falso
    inicializa i ← 0
    Mientras i > 0 y i <15 hacer
        Mientras tam_cadena > 3 y tam_cadena < 6 hacer
            Mientras j > 1 y j < tam_cadena - 1 hacer
                min_t ← mínimo de las marcas de tiempo
            fin
            Mientras j > tam_cadena y j < 15 hacer
                max_t ← máximo de las marcas de tiempo
                Si max_t ≥ min_t entonces
                    no_es_esquina ← verdadero
                    sale bucle
                fin
                Si no_es_esquina es falso entonces
                    es_esquina ← verdadero
                    sale bucle
                fin
            fin
            Si es_esquina es verdadero entonces
                sale bucle
            fin
        fin
    fin
    Si es_esquina es verdadero entonces
        es_esquina ← falso Mientras i > 0 y i <19 hacer
            Mientras tam_cadena > 4 y tam_cadena < 8 hacer
                Mientras j > 1 y j < tam_cadena - 1 hacer
                    min_t ← mínimo de las marcas de tiempo
                fin
                Mientras j > tam_cadena y j < 15 hacer
                    max_t ← máximo de las marcas de tiempo
                    Si max_t ≥ min_t entonces
                        no_es_esquina ← verdadero
                        sale bucle
                    fin
                    Si no_es_esquina es falso entonces
                        es_esquina ← verdadero
                        sale bucle
                    fin
                fin
                Si es_esquina es verdadero entonces
                    sale bucle
                fin
            fin
        fin
    fin
    retornar es_esquina
fin
```

Cómo puede apreciarse en el Algoritmo 1, son necesarios varios bucles anidados que recorren todos los valores con las marcas de tiempo de los eventos que rodean al que se procesa. Trataremos de encontrar una solución más eficiente que aproveche el paralelismo de las FPGAs.

### 5.2.2 Implementación en FPGAs

Para poder procesar los eventos empleando el algoritmo FAST, es ineludible, primero, leer los datos de memoria. Con cada dato leído podríamos implementar algún algoritmo que sea capaz de realizar operaciones de forma que, al leer el último dato, se reconozca, en ese mismo ciclo de reloj, si existe o no una cadena. Sin embargo, si bien se ha probado esta forma, para este trabajo presento un enfoque diferente, en el que primero almaceno las marcas de tiempo de los eventos que rodean al que se procesa y a continuación hacemos operaciones sobre ellas. De esta forma se consigue ahorrar recursos, aunque aumentando la latencia y el tiempo de procesamiento, como se expondrá en el próximo capítulo.

Se ha diseñado una máquina de estados que permite que al recibir un evento se siga la secuencia representada en la Fig. 5.4: primero, se leen de memoria BRAM los valores asociados a la circunferencia interior y se almacenan en un vector; segundo, se leen los valores de la circunferencia exterior, que se almacenarán en otro vector; tercero, comprobamos si en el vector de la circunferencia de radio 3 existe o no una cadena de valores contiguos que sean mayores que el resto de datos; cuarto, buscamos una cadena, pero ahora en el vector de los datos asociados a la circunferencia de radio 4. Si en ambos vectores existe alguna cadena, entonces se considera que el evento es una esquina.

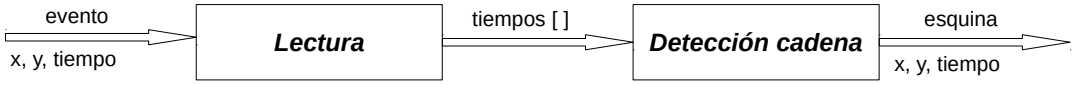



**Figura 5.3** Diagrama básico de funcionamiento.

Aprovechando la capacidad de las FPGAs de implementar bloques lógicos que operen en paralelo, las etapas (1) y (4) así como las (2) y (3) pueden ejecutarle a la vez, de forma que, tras leer los datos de la circunferencia interna, se detecta si hay o no cadenas en ella al tiempo que se leen los datos de la circunferencia externa. En la siguiente fase, mientras se procesa el vector con las marcas de tiempo de la cadena externa, se empiezan a leer los datos de la circunferencia interna del siguiente evento. Dado que los eventos se reciben en paquetes, resulta conveniente procesar los datos en una *pipeline*, como se ha expuesto.

#### Etapas 1 y 3: Lectura

Para que el procesamiento sea lo más rápido posible, se ha almacenado en memoria BRAM (*Block* RAM) la marca de tiempo del último evento recibido en la posición de cada píxel. La memoria debe tener, pues, una profundidad de $h * w$ palabras, donde $h$ es la altura del sensor, y $w$ el ancho.

El bloque de BRAM es generado a través de una IP (*Intellectual Property*) facilitada por *Xilinx*. El bloque de memoria permite leer hasta dos datos cada ciclo de reloj, con una latencia de 2 ciclos de reloj; es decir, cuando establecemos en el puerto correspondiente la dirección de la que queremos leer el dato (*Addr* en la Fig. 5.5), éste tarda dos ciclos de reloj en estar disponible en el puerto de salida del dato (*Rd Data*). Siendo $(x, y)$ la posición del evento, la dirección de memoria se calculará de la forma:

$$Addr = y * w + x$$

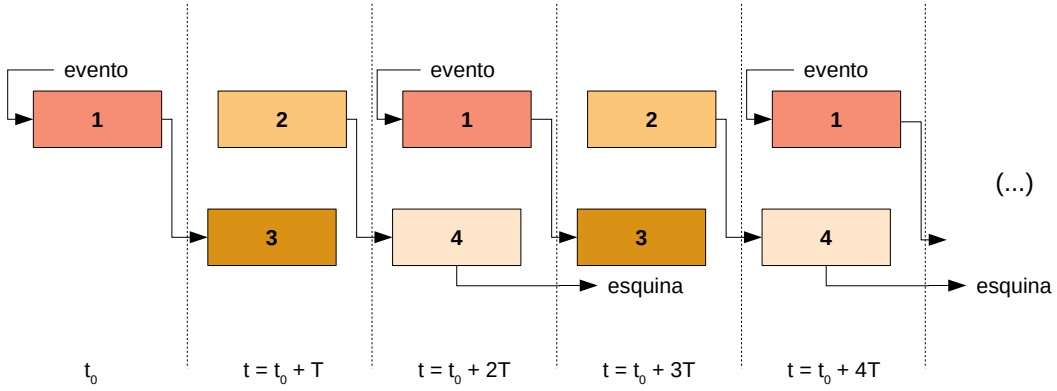



**Figura 5.4** El proceso incluye: (1) Lectura de datos de la circunferencia interna; (2) Lectura de datos de la circunferencia externa; (3) Detección de cadena en la circunferencia interna; (4) Detección de cadena en la circunferencia externa. Cada fase dura T ciclos de reloj.

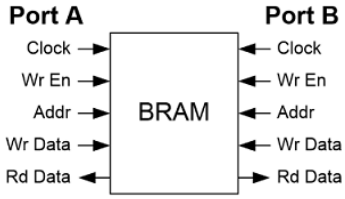



**Figura 5.5** Block RAM. Figura tomada de NandLand. Accedido el 31 de agosto de 2024.

Teniendo esto en consideración, buscamos, primero, guardar en un vector los datos asociados a la circunferencia interna, para después enviarlo al bloque que detecta cadenas; y, tras esto, leemos los datos de la circunferencia externa para después procesarlo. El proceso es:

- Duración de 2 ciclos de reloj: preparación de las cuatro primeras direcciones de memoria asociadas a la circunferencia interior, dos por cada ciclo de reloj.
- Duración de 6 ciclos de reloj: preparación de las 12 siguientes direcciones asociadas a la circunferencia interior y lectura de los 12 primeros datos de la circunferencia interior.
- Duración de 2 ciclos de reloj: lectura de los 4 últimos datos de la circunferencia interior y preparación de las 4 primeras direcciones de la cadena exterior. Mandamos el vector al bloque que detecta cadenas.
- Duración de 8 ciclos de reloj: preparación de las 16 siguientes direcciones asociadas a la circunferencia exterior y lectura de los 16 primeros datos de la circunferencia exterior.
- Duración de 2 ciclos de reloj: lectura de los 4 últimos datos de la circunferencia exterior. Mandamos el vector al bloque que detecta cadenas.
- Duración de 1 ciclo de reloj: escritura de la marca de tiempo del evento recibido en memoria. Esperamos al siguiente evento.

#### Etapas 2 y 4: Detección de cadenas

En las etapas 2 y 4 recibiremos del bloque de lectura un vector con las marcas de tiempo de los eventos que rodean al evento recibido según el patrón descrito anteriormente. Teniendo dicho vector como entrada, debemos detectar la presencia de cadenas.

Se han probado diferentes implementaciones, todas ellas adaptadas a las cualidades de los circuitos digitales, pero por motivos que se expondrán en el Capítulo 6, se ha optado por una que difiere notablemente de la implementación en procesadores de [29]. Ilustraremos esta implementación para la etapa 2, en la que buscamos cadenas en la circunferencia de radio 3.

En *VHDL*, que es el lenguaje de descripción *hardware* que se ha empleado para programar la FPGA, existe un tipo de dato llamado *record*, similar a una estructura en otros lenguajes de programación secuencial. En nuestro código se ha definido un *record* con tantas componentes como datos se quieran evaluar para detectar cadenas: 16 para la circunferencia interior y 20 para la exterior. Cada componente contendrá el valor con la marca de tiempo y el índice asociado en la circunferencia, como se muestra en la Fig. 5.6 .

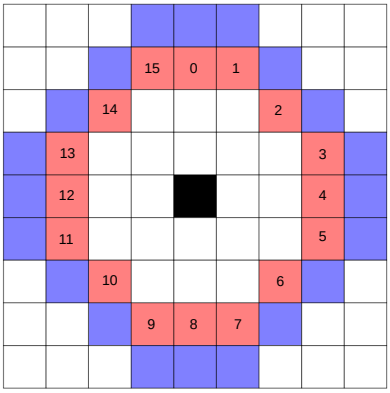
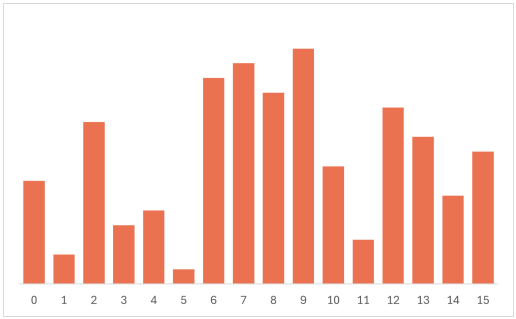
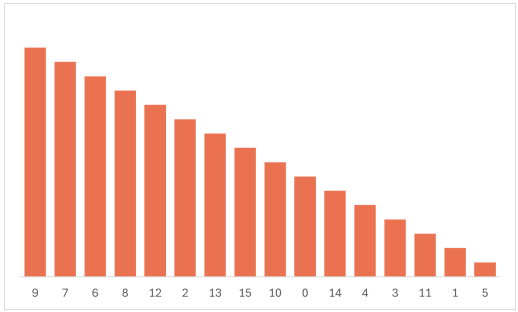



**Figura 5.6** Circunferencia interna indexada. Figura adaptada de [29].

Teniendo estos datos, primero los ordenaremos de mayor a menor junto con sus índices. Con los datos ordenados, solo necesitaremos el vector de índices para continuar.




**(a)** Vector con marcas de tiempo.




**(b)** Vector con marcas de tiempo ordenado.

**Figura 5.7** Representación de posibles valores de marcas de tiempo de la circunferencia interna.

Para realizar el ordenamiento no hemos hecho uso de algoritmos ampliamente conocidos en la programación secuencial como *bubble sort*, *quick sort*, o similares; sino que hemos empleado una red de ordenamiento (en inglés: *sorting network*), que hace esta implementación viable.

Una red de ordenamiento es un concepto empleado en computación que representa una serie de cables que transportan valores y que se conectan por pares a través de comparadores, que permiten la reorganización de los datos. En la Fig. 5.8 se presenta un ejemplo en el que se ordenan 4 datos. La red es igual para ordenar los datos de mayor a menor o de menor a mayor, variando únicamente el signo de los comparadores, representados por las líneas verticales.

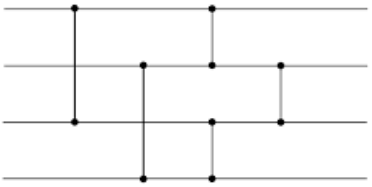


**Figura 5.8** Red de ordenamiento de 4 elementos. Figura tomada de Wikipedia. Accedido el 31 de agosto de 2024.

Existen diversas redes de ordenamiento, como *batcher odd-even mergesort*, *shell sortpairwise sorting network*, o *bitonic sort*, todas ellas con la misma carga computacional. En este trabajo se ha optado por *bitonic sort*, el más empleado en GPUs.

En la Fig. 5.9 se ha representado una red bitónica (*bitonic* en inglés) que permite ordenar los 16 valores que componen la circunferencia interna. Del mismo modo, la Fig. 5.10 muestra la red bitónica de 20 componentes. Las redes de ordenamiento son recursivas; se pueden ordenar vectores de $2^n$ valores a partir de redes de $2^{n-1}$ valores. Por este motivo, la red de 20 componentes, que se ha obtenido eliminando las comparaciones innecesarias de una red de 32, contiene a la red de 16 líneas. En las Figs. 5.9 y 5.10, cada una de las fases, numeradas en cifras romanas, se ejecuta en un ciclo de reloj.

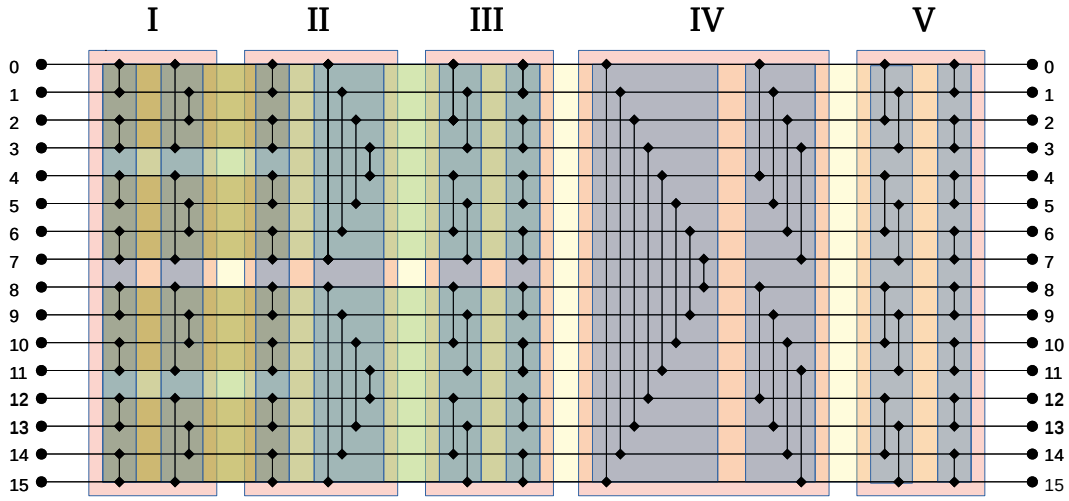



**Figura 5.9** Red de ordenamiento bitónica con 16 líneas.

Una vez los datos están ordenados de mayor a menor a la salida de la red de ordenamiento como en la Fig. 5.7, emplearemos los índices para evaluar la presencia de cadenas.

En el Algoritmo 2 se describe la función que se ha empleado, teniendo como parámetro de entrada el vector con los índices ordenados y como valor de salida el resultado (= 1 si hay cadena). Esta función sólo detectará la cadena si está en el rango definido por *min_tam_cadena* y *max_tam_cadena*, que toman valor de 3 y 6, respectivamente, para la circunferencia interior, y 4 y 8 para la exterior.

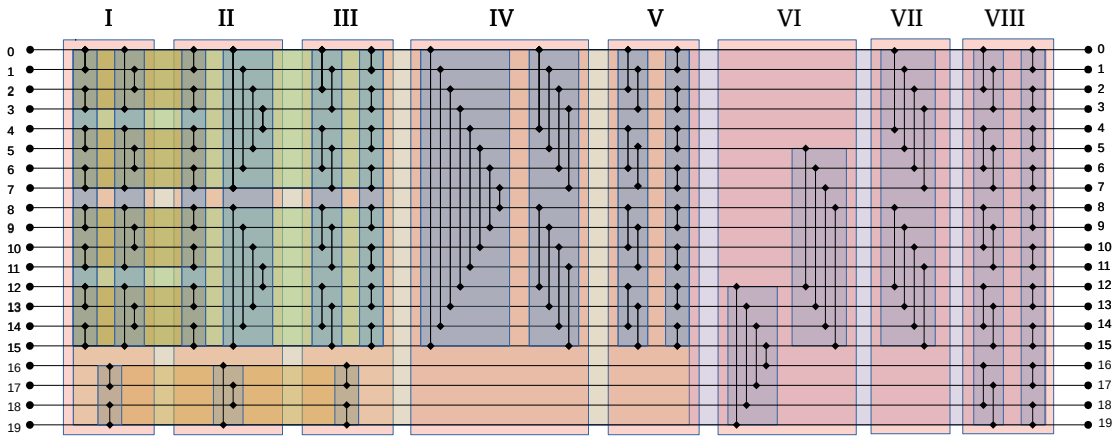



**Figura 5.10** Red de ordenamiento bitónica con 32 líneas.

**Algoritmo 2:** Pseudocódigo para la función que reconoce si hay cadena en el vector ordenado

```
Input: vector_idx
Output: resultado
Función detecta_cadena:
    variable resultado ← 0 variable mínimo ← vector_idx(0) variable
      máximo ← vector_idx(0) inicializa i ← 0
    Mientras i > 0 y i < max_tam_cadena hacer
        Si vector_idx(i) > máximo entonces
          | máximo ← vector_idx(i)
        fin
        SinoSi vector_idx(i) < mínimo entonces
          | mínimo ← vector_idx(i)
        fin
        Si i ≤ min_tam_cadena − 1 y máximo − mínimo = i entonces
          | resultado ← 1
        fin
        i ← i + 1;
    fin
    retornar resultado;
fin
```

Sin embargo, esta función no resulta válida si dos de los valores de la cadena están en los índices mínimo y máximo, 0 y 15 o 0 y 19, según la circunferencia que se esté evaluando. Si éste es el caso, la cadena es detectada con el algoritmo 3.

Aunque *VHDL* es un lenguaje de descripción *hardware*, permite definir funciones expresadas de forma secuencial, como los algoritmos antes citados. En este caso, el programador no tiene control a nivel RTL (*register transfer level*), por lo que pueden resultar circuitos combinacionales de gran tamaño, pudiendo incumplir ciertas restricciones temporales. Esto ha ocurrido al intentar sintetizar la segunda función, que se ejecutaría en un único ciclo de reloj; por lo que, por ahora, solo se detectarán las cadenas que permita el Algoritmo 2.

En el próximo capítulo se harán más comentarios relacionados con las restricciones temporales y los recursos empleados.

**Algoritmo 3:** Pseudocódigo para la función que reconoce si hay cadena en el vector ordenado y que pase por la primera o última posición de la circunferencia

```
Input: vector_idx
Output: resultado
Función detecta_cadena:
    variable resultado ← 0 variable mínimo ← 0 variable máximo ←
     longitud_caden- 1 inicializa i ← 0
    Mientras i > 0 y i < max_tam_cadena hacer
        Si vector_idx(i) < longitud_cadena/2 entonces
            Si vector_idx(i) > mínimo entonces
                mínimo ← vector_idx(i)
            fin
        fin
        SinoSi vector_idx(i) < máximo entonces
            máximo ← vector_idx(i)
        fin
        Si i ≥ min_tam_cadena − 1 y mínimo + longitud_caden −
         máximo = i entonces
            resultado ← 1
        fin
        i ← i + 1;
    fin
    retornar resultado;
fin
```

## 5.3 Velocidad de procesamiento

Con esta implementación de algoritmo FAST se podrán procesar eventos a una tasa constante de $200ns/evento$, o lo que es lo mismo: $5Meventos/s$.

En [26] se presentan los resultados de implementar el mismo algoritmo en una *Zynq 7Z007S*, un sistema similar al que empleamos en nuestro trabajo, integrando también CPU y FPGA. En ese artículo se determina que el procesamiento de esquinas empleando FAST en un ordenador con una CPU de cuatro núcleos a 2.3GHz, tarda entre 0.6 y 30 *us* en procesar cada evento; es decir, es entre 3 y 150 veces más lento que nuestra implementación.

Sin embargo, en [26] también emplean un bloque de procesamiento para la programación de la FPGA, aunque éste es creado con *Vitis HLS*, que permite generar el bloque a nivel de registros a partir de código de alto nivel (C/C++), lo que resulta, en principio, en una solución menos eficiente que la obtenida con lenguajes de descripción *hardware* como *VHDL* o *Verilog*. De esta forma, han conseguido que su bloque procese datos entre 100 y 200 *ns*, dependiendo del entorno del evento.

En su trabajo no especifican a qué se debe la dependencia del entorno: si cada evento tarda en procesarse 200 ns salvo si éste está en los bordes (si el evento ocurre en las cuatro primeras o últimas filas o columnas, no se procesa, tan solo se escribe en memoria), lo que tarda 100 ns, entonces

el nuestro es más eficiente, ya que si el evento está en los bordes con nuestra implementación tarda 10 *ns*; y si es porque primero evalúa si existe o no una cadena en la circunferencia interna (necesariamente la interna), y si no existe rechaza el evento como esquina, el suyo resulta más eficiente.

La segunda de las dos opciones es posible, aunque implica que son capaces de detectar si hay o no cadena en la circunferencia interna en sólo 10 ciclos de reloj (emplean una frecuencia de reloj de $100MHz$ en la FPGA, como nosotros). Recordamos que en cada ciclo de reloj sólo se pueden leer dos datos de memoria, por lo que se necesitan exactamente 10 ciclos de reloj para leer todos los datos de la circunferencia interna (los 2 primeros ciclos preparan las direcciones de los 4 primeros datos), de forma que el bloque debe implementar la lógica necesaria para determinar la presencia de cadena en el último ciclo de reloj.

Este fue el primer enfoque que se siguió para la programación del módulo de procesamiento, pero su implementación no cumplía las restricciones temporales, quizá por emplear una cámara con mayor resolución que la del artículo mencionado o con una mayor precisión de las marcas de tiempo.

En todo caso, en [26] no mencionan cómo se produce la tranferencia de datos entre PL o PS, por lo que su implentación puede estar limitada.

# 6 Evaluación

En este capítulo veremos los materiales que se han empleado para poner el sistema en funcionamiento, y, además, las simulaciones y experimentos que se han realizado.

## 6.1 Placa de desarrollo

La placa de desarrollo que se ha utilizado es la *AMD Kria™ KV260 Vision AI Starter Kit* (Fig. 6.1). Esta placa contiene el K26 SOM (*system-on-module*), que integra una *AMD Zynq™ UltraScale+™ MPSoC*, así como una placa portadora con otros periféricos y componentes. La *Kria KV260* está especialmente dirigida a aplicaciones que emplean aprendizaje máquina en visión.

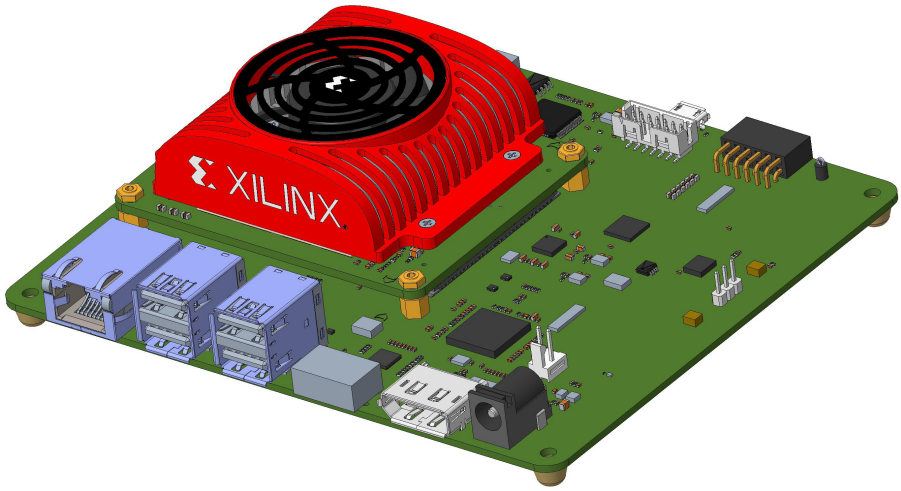



**Figura 6.1** *Kria KV260*. Figura tomada de AMD. Accedido el 31 de agosto de 2024.

Uno de los motivos por los que se escogió esta placa, además de su reducido precio en comparación con otras que integran la misma arquitectura, es el puerto *USB 3.0* que incluye, lo que lo hace compatible con la cámara que emplearemos. Otras de las interfaces presentes en la placa (ver Fig. 6.2) también nos serán de ayuda, como el puerto para la tarjeta *microSD* que contiene la imagen del sistema operativo; el Ethernet, necesario para la transmisión de *sockets*; o el *JTAG/UART*, empleado para el primer arranque del sistema operativo y para labores de depuración.

Antes de conectar la cámara, es conveniente probar que el sistema funciona como se espera mediante simulaciones. Las simulaciones facilitan la depuración de errores, y han sido fundamentales

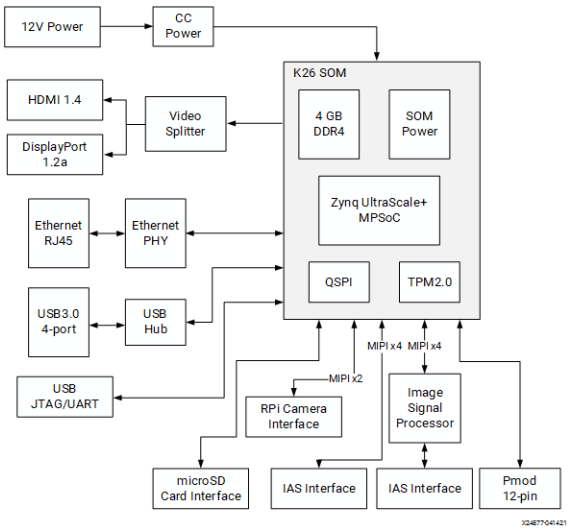



**Figura 6.2** Periféricos de la *Kria KV260*. Figura tomada de AMD. Accedido el 31 de agosto de 2024.

en la programación del módulo de procesamiento de eventos (*ev_cam_coner_fast*), descrito en el Capítulo 5.

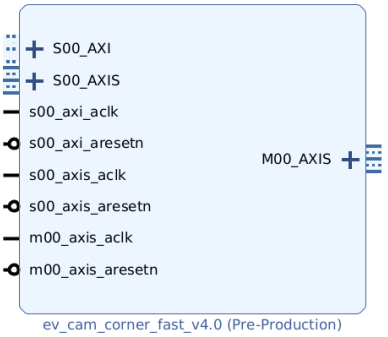



**Figura 6.3** Bloque procesamiento de eventos generado en *Vivado*.

## 6.2 Simulación

Para la simulación del bloque de procesamiento de eventos (no de todo el sistema) se ha diseñado un banco de pruebas en el que se leerán de un archivo los datos de eventos, que vendrán en el formato que se indica en la Fig. 3.5, para ser procesados. Estos datos leídos entrarán a través del *stream* de datos por el canal **S00_AXIS** y, cuando se alcance un número determinado de esquinas, saldrán a través del canal **M00_AXIS**, también en forma de *stream*, ver Fig. 6.3.

Los datos de eventos no se han introducido de forma arbitraria en el archivo: primero, mandaremos datos que se corresponderán con las todas las posibles posiciones del sensor, y cada uno de estos datos contendrá una marca de tiempo aleatoria; después, se mandarán datos que sabemos que, si el algoritmo se ha implementado correctamente, resultarán identificados como esquinas.

De esta forma, si queremos que un dato sea reconocido como esquina, antes debemos enviar los datos correspondientes a los eventos que lo rodean, añadiendo cadenas con valores consecutivos

que sean mayores que el máximo de los valores aleatorios antes introducidos. Si los valores no fueran aleatorios, habría muchos falsos positivos al darse gran presencia de ceros en la matriz de marcas de tiempo, complicando la depuración.

En la Fig. 6.4 se muestra el resultado de la simulación en *Vivado*, con la representación de dos de las señales digitales internas del bloque de procesamiento (hay muchas más, que por claridad no se han representado). La señal **new_event** toma valor digital alto cuando se recibe un nuevo evento a la entrada del bloque, y la señal **found_corner** se ”activa” cuando el evento de entrada es una esquina. Como se puede apreciar en la Fig. 6.4, se mandan bloques de 16 datos (eventos de la circunferencia interna) y de 20 (de la externa), y a continuación el evento central, que debe activar la señal **found_corner**.

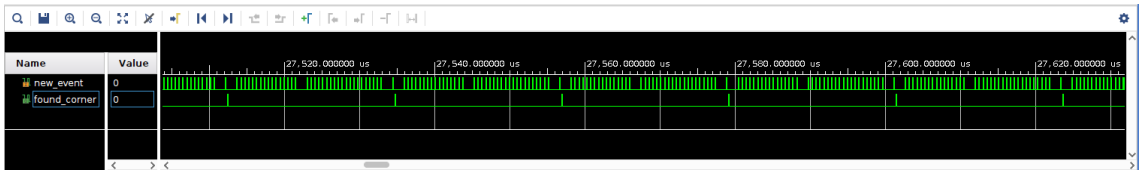



**Figura 6.4** Simulación del detector de esquinas.

En la Fig. 6.4 todas las esquinas se han identificado correctamente; sin embargo, en la Fig. 6.5, no se reconocen todas las esquinas. En esta última figura, se ha aumentado la escala temporal, para poder visualizar más eventos. La separación entre las activaciones de la señal **found_corner** deberían ser constantes y aproximadamente igual a 200*ns*.

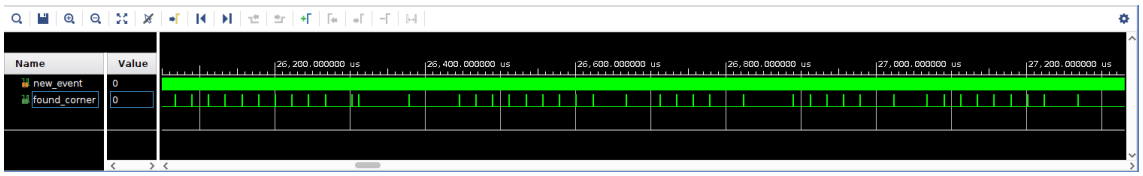



**Figura 6.5** Simulación del detector de esquinas.

Como se comentó en el Capítulo 5, había dos funciones que debían ejecutarse para detectar todas las cadenas posibles, aunque, por problemas de cumplimiento de requisitos temporales, sólo podíamos ejecutar una de las dos. De esta forma, las esquinas que no se detectan son las que tienen la cadena en la parte superior de la circunferencia, que contiene el primer y último índice asociado a los datos que forman el vector de marcas de tiempos en el que se detecta la cadena.

El resto del sistema no se ha simulado, aunque para la corrección de errores asociados a la DMA se ha empleado un bloque ILA, mencionado en el Capítulo 4, que permite ver señales de la FPGA en tiempo real para depurar el código.

## 6.3 Recursos empleados

En las Figs. 6.6 y 6.7 se muestran los recursos de la FPGA que consume el módulo de procesamiento de eventos. Este bloque emplea un 7.42 % de las LUTs (*Look Up Table*, y un 56.94 % de la memoria BRAM. Dado que la resolución de la cámara que emplearemos es de 260*x*346 y la marca de tiempo de los eventos ocupa 32 bits, la memoria que contiene los últimos eventos asociados a cada posición

del sensor ocupa unos 360*KB*, que equivale a 80 bloques BRAM. Los otros dos bloques son de la memoria en la que se almacenan las esquinas antes de mandarlas al PS a través de DMA.

Debido a que la *Kria* sólo tiene 144 bloques, no podemos emplear dos bloques de memoria diferentes para cada polaridad de los eventos, como en [29].

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| CLB LUTs* | 8691 | 0 | 0 | 117120 | 7.42 |
| LUT as Logic | 8691 | 0 | 0 | 117120 | 7.42 |
| LUT as Memory | 0 | 0 | 0 | 57600 | 0.00 |
| CLB Registers | 1941 | 0 | 0 | 234240 | 0.83 |
| Register as Flip Flop | 1941 | 0 | 0 | 234240 | 0.83 |
| Register as Latch | 0 | 0 | 0 | 234240 | 0.00 |
| CARRY8 | 230 | 0 | 0 | 14640 | 1.57 |
| F7 Muxes | 634 | 0 | 0 | 58560 | 1.08 |
| F8 Muxes | 83 | 0 | 0 | 29280 | 0.28 |
| F9 Muxes | 0 | 0 | 0 | 14640 | 0.00 |

**Figura 6.6** Recursos empleados por el módulo de procesamiento de eventos.

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Block RAM Tile | 82 | 0 | 0 | 144 | 56.94 |
| RAMB36/FIFO* | 82 | 0 | 0 | 144 | 56.94 |
| RAMB36E2 only | 82 | | | | |
| RAMB18 | 0 | 0 | 0 | 288 | 0.00 |
| URAM | 0 | 0 | 0 | 64 | 0.00 |

**Figura 6.7** Memoria empleada por el módulo de procesamiento de eventos.

El diseño completo requerirá de los recursos mostrados en las Figs. 6.8 y 6.9. La memoria adicional que se emplea es de las FIFOs.

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| CLB LUTs | 19215 | 0 | 0 | 117120 | 16.41 |
| LUT as Logic | 17417 | 0 | 0 | 117120 | 14.87 |
| LUT as Memory | 1798 | 0 | 0 | 57600 | 3.12 |
| LUT as Distributed RAM | 676 | 0 | | | |
| LUT as Shift Register | 1122 | 0 | | | |
| CLB Registers | 17553 | 0 | 0 | 234240 | 7.49 |
| Register as Flip Flop | 17553 | 0 | 0 | 234240 | 7.49 |
| Register as Latch | 0 | 0 | 0 | 234240 | 0.00 |
| CARRY8 | 353 | 0 | 0 | 14640 | 2.41 |
| F7 Muxes | 760 | 0 | 0 | 58560 | 1.30 |
| F8 Muxes | 131 | 0 | 0 | 29280 | 0.45 |
| F9 Muxes | 0 | 0 | 0 | 14640 | 0.00 |

**Figura 6.8** Recursos empleados por todo el diseño.

En la Fig. 6.10 se muestran estos recursos que exige nuestra aplicación en la *Zynq UltraScale+ MPSoC*. El rectángulo amarillo representa los procesadores, y el resto el tejido lógico, incluyendo memoria BRAM y puertos de la FPGA.

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Block RAM Tile | 109 | 0 | 0 | 144 | 75.69 |
| RAMB36/FIFO* | 107 | 0 | 0 | 144 | 74.31 |
| RAMB36E2 only | 107 | | | | |
| RAMB18 | 4 | 0 | 0 | 288 | 1.39 |
| RAMB18E2 only | 4 | | | | |
| URAM | 4 | 0 | 0 | 64 | 6.25 |

**Figura 6.9** Memoria empleada por todo el diseño.

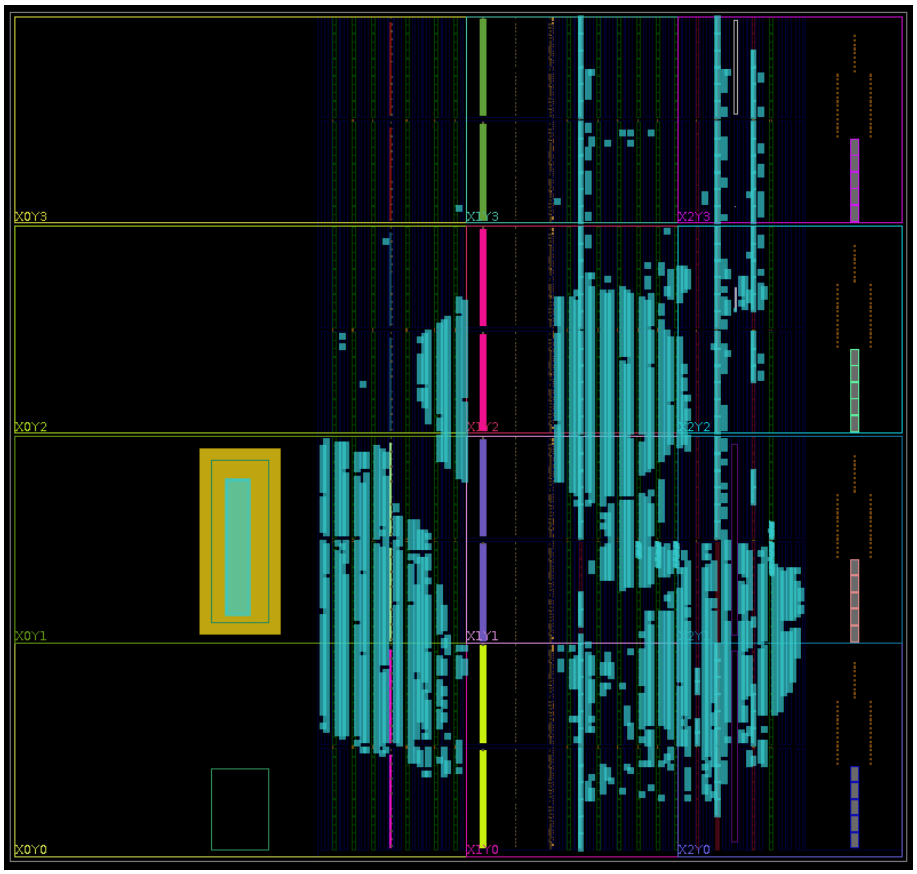



**Figura 6.10** Resultado de la implementación.

Un aspecto importante a tener en consideración para la programación de las FPGAs es el tiempo que tardan las señales en propagarse por el circuito. Si el tiempo que requiere la señal en llegar de un registro a otro es mayor que el periodo de la señal de reloj, el circuito que se implementa no estará sincronizado, dando lugar así a comportamientos imprevistos. En la Fig. 6.11 se pueden ver los resultado del análisis de tiempo realizado en nuestro circuito. *Worst Negative Slack* representa la mínima holgura que hay entre el periodo de la señal de reloj y el tiempo que tarda una señal de llegar de un registro a otro. Si este valor es negativo, no se cumplen las restricciones temporales, y el circuito no funcionará como es esperado.

Un problema que presenta nuestra aplicación es la gran cantidad de memoria BRAM que emplea; y es que la lógica debe ser capaz de acceder a todas las celdas de memoria, que se encuentran distribuidas en el circuito integrado, haciendo que el retraso debido a la red se incremente considerablemente.

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.029 ns | Worst Hold Slack (WHS): | 0.010 ns | Worst Pulse Width Slack (WPWS): | 3.500 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 55294 | Total Number of Endpoints: | 55278 | Total Number of Endpoints: | 20827 |

**All user specified timing constraints are met.**

**Figura 6.11** Resultado del análisis de tiempo.

En este tiempo que tarda la señal en transmitirse entre dos puntos del circuito se consideran tanto el producido por la red de conexionado de la FPGA como el producido por la misma lógica. Y si bien la profundidad de memoria es un problema, lo que imposibilita el uso de la función que reconoce las cadenas que cruzan por los índices primero y último, como se comentó anteriormente, es también el retraso producido por la lógica que genera el sintetizador, que resulta similar que el retraso producido por los recursos de interconexión.

En la Fig. 6.12 se muestran las rutas para las que se produce menor holgura. La ruta señalada surge al emplear la función que reconoce las cadenas (la que sí hemos podido implementar). Como podemos ver, al no trabajar esta función sobre las marcas de tiempo de los eventos, que tienen profundidad de 32 bits, requiriendo más LUTs y, por tanto, mecanismos de enrutado adicionales; el retraso producido por la lógica es considerablemente mayor que para el resto de rutas, que requieren menor lógica pero más conexiones.

| Name | Slack ^1 | Levels | High Fanout | From | Total Delay | To | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 0.029 | 39 | 35 | block_design_...[0]_replica/C | 9.796 | block_design...corner_reg/D | 5.229 | 4.567 | 10.0 | clk_pl_0 |
| Path 2 | 1.095 | 0 | 80 | block_design...drb_reg[2]/C | 7.982 | block_desig...RBWRADDR[2] | 0.116 | 7.866 | 10.0 | clk_pl_0 |
| Path 3 | 1.141 | 0 | 80 | block_design...drb_reg[2]/C | 7.930 | block_desig...RBWRADDR[2] | 0.116 | 7.814 | 10.0 | clk_pl_0 |
| Path 4 | 1.224 | 0 | 80 | block_design...drb_reg[2]/C | 7.859 | block_desig...RBWRADDR[2] | 0.116 | 7.743 | 10.0 | clk_pl_0 |
| Path 5 | 1.262 | 0 | 80 | block_design...dra_reg[0]/C | 7.951 | block_desig...RARDADDR[0] | 0.114 | 7.837 | 10.0 | clk_pl_0 |
| Path 6 | 1.342 | 0 | 80 | block_design...drb_reg[2]/C | 7.750 | block_desig...RBWRADDR[2] | 0.116 | 7.634 | 10.0 | clk_pl_0 |
| Path 7 | 1.397 | 0 | 80 | block_design...dra_reg[0]/C | 7.822 | block_desig...RARDADDR[0] | 0.114 | 7.708 | 10.0 | clk_pl_0 |

**Figura 6.12** *Worst Negative Slack*.

## 6.4 Experimentos

Una vez comprobado que el funcionamiento del sistema es el esperado mediante simulación, integraremos todos los componentes, que nos permitirán visualizar directamente los resultados.

La cámara de eventos que hemos usado es la *Davis346*, de la empresa *iniVation*. Esta cámara tiene una resolución de $346x260$, un rango dinámico de $120dB$, un ancho de banda de $12Meventos/segundo$ y un consumo de $180mA$ a $5V$ DC. Además, integra una IMU (Unidad de Medición Inercial) y un sensor que permite obtener imagen en escala de grises, aunque no emplearemos estas dos características en nuestro trabajo.

Sin embargo, para poder depurar nuevos algoritmos sin necesitad de tener una cámara disponible, se ha programado también un emulador, con el que igualmente podremos ver los eventos procesados por el sistema, aunque estos en vez de proceder de una cámara, se tomarán de una base de datos.

Para comunicarnos con la *kria* debemos conectar el cable Ethernet, como se puede ver en la Fig. 6.14, para así poder usar *SSH* y acceder al terminal de la placa de desarrollo desde nuestro portátil. Siempre que arranquemos el sistema debemos, primero, cargar el módulo kernel y la aplicación *hardware*; y, hecho esto, podremos ejecutar las aplicaciones de usuario.

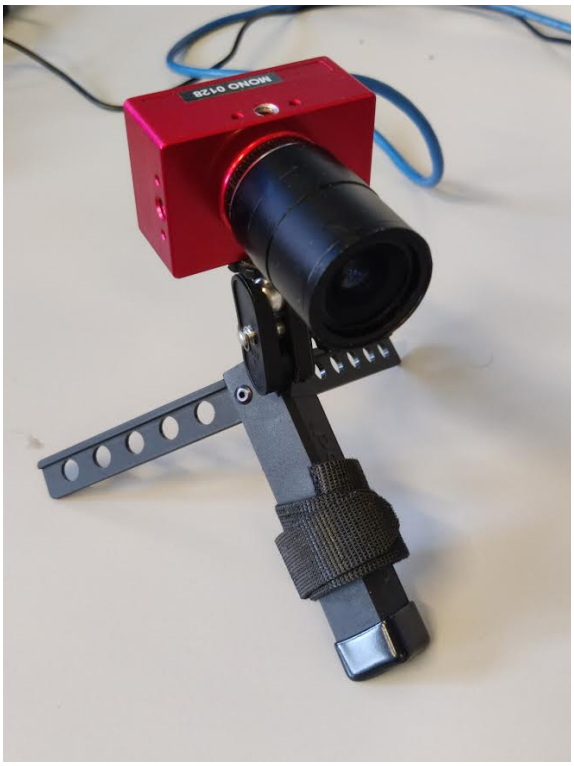



**Figura 6.13** Cámara de eventos *Davis346*.

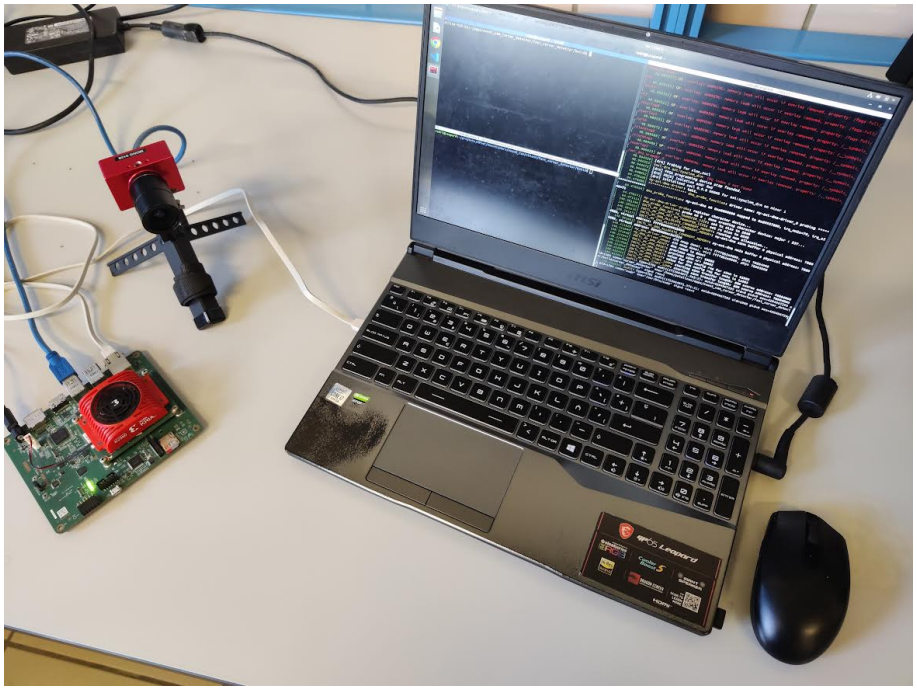


**Figura 6.14** Montaje del sistema para los experimentos.

En el terminal superior izquierdo (ver Fig. 6.15) hemos ejecutado la aplicación de usuario de la *kria*, y en el derecho la aplicación de usuario en nuestro portátil, que permite la visualización. En el terminal derecho vemos los mensajes que se generan desde el módulo kernel. En rojo se resalta la llegada una interrupción producida por el recibimiento a través de la DMA de un nuevo paquete de esquinas.

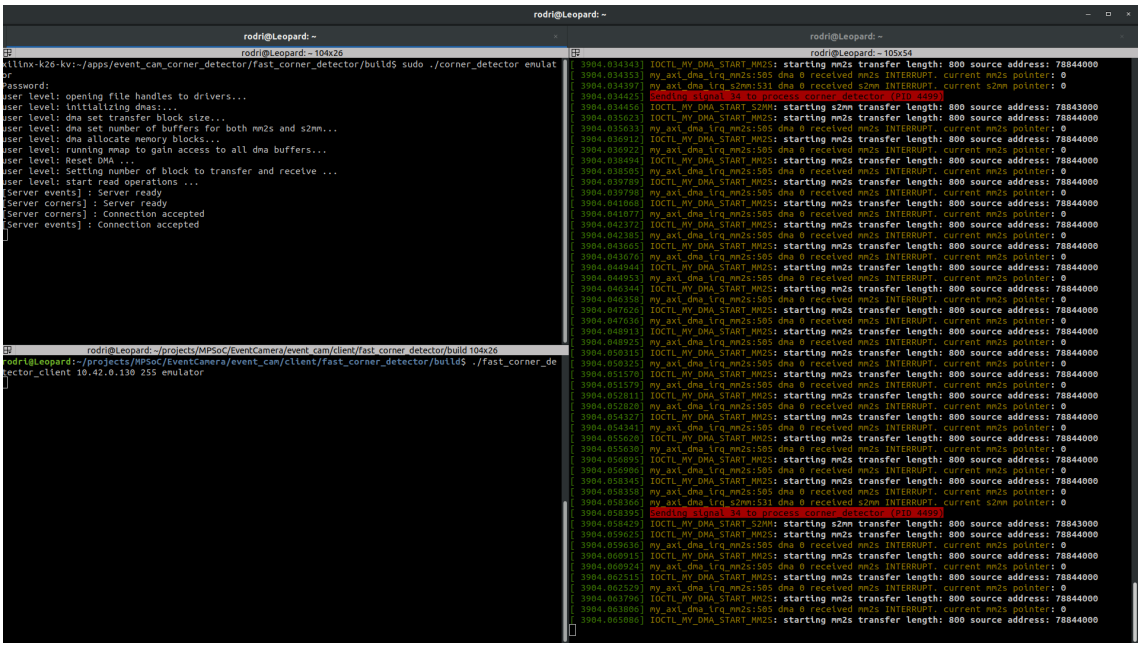



**Figura 6.15** Ejecución de las aplicaciones.

El mismo programa se emplea para hacer funcionar el sistema tanto con la cámara con el emulador, aunque hay que pasarle como parámetro cuando lo ejecutemos en el terminal de dónde se obtendrán los eventos.

Al ejecutar los programas emergen dos ventanas en las que se muestran imágenes generadas por todos los eventos y, de forma separada, por las esquinas. En la Fig. 6.16 se puede ver una captura de lo que se obtiene al trabajar con el emulador.

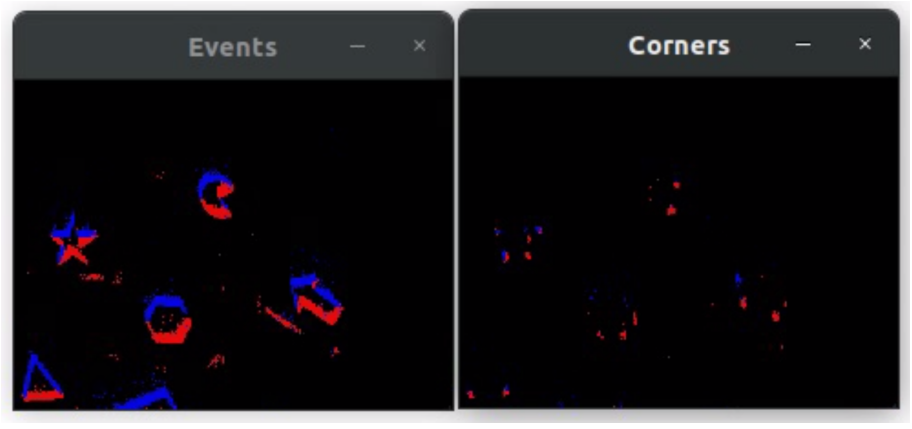



**Figura 6.16** Resultado con el emulador.

En la imagen con los resultados puede apreciarse cómo no se detectan correctamente las esquinas superiores de las figuras, que coinciden, en este caso, con eventos con polaridad negativa, representados en azul. Como comentamos en el Capítulo 5, para poder determinar que un evento sea una esquina, han de detectarse cadenas; pero si éstas cruzan por el primer y último índice del vector de

marcas de tiempo (ver Fig. 6.17), no se reconoce como esquina.

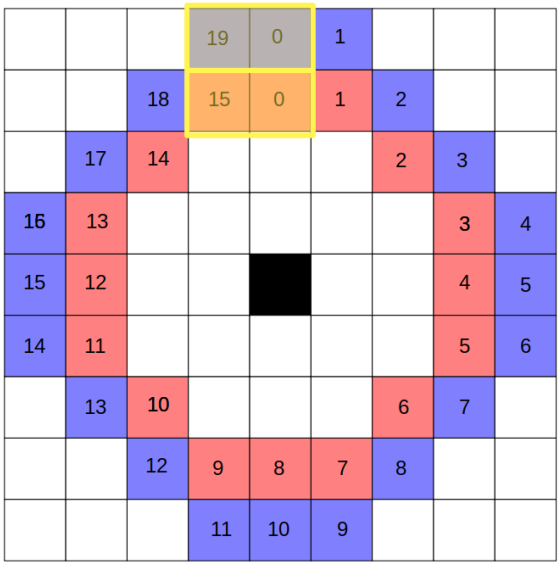



**Figura 6.17** Circunferencias numeradas. Una cadena que contenga las dos posiciones marcadas en amarillo no se reconocerá. Figura adaptada de [29].

Si nos fijamos en el triángulo de la esquina inferior izquierda de la Fig. 6.16, vemos como en el vértice superior no se detecta la esquina, al igual que ocurre con otros vértices con la misma orientación en las diferentes figuras.

En la Fig. 6.18 se muestran las cadenas que produciría el vértice superior del triángulo. Al cruzar ambas cadenas por la primera y última posición del vector de forma simultánea (aunque con una cadena basta), no detectaremos esta esquina. Se representan dos pares de líneas que se corresponden con el mismo vértice en dos momentos diferentes de su movimiento vertical. Sabemos que se mueve de esta forma (concretamente, de manera descendente) por los colores en la Fig. 6.16, que representan la polaridad.

Con la limitación de no poder reconocer ciertas esquinas, el procesamiento se realiza a una tasa constante de 200$ns$, aunque se podría hacer que el procesamiento sea más eficiente en términos de precisión (detectando todas las esquinas que permite el algoritmo FAST) a costa de aumentar en varios ciclos de reloj el tiempo de procesamiento.

Accediendo a este enlace se puede ver un vídeo con la detección de esquinas empleando el emulador. Como se comentó anteriormente, cada imagen se actualiza cuando se reciben 256 esquinas desde el bloque de procesamiento. En la imagen de eventos se incluyen todos los eventos que han sido necesarios procesar para detectar esas 256 esquinas.

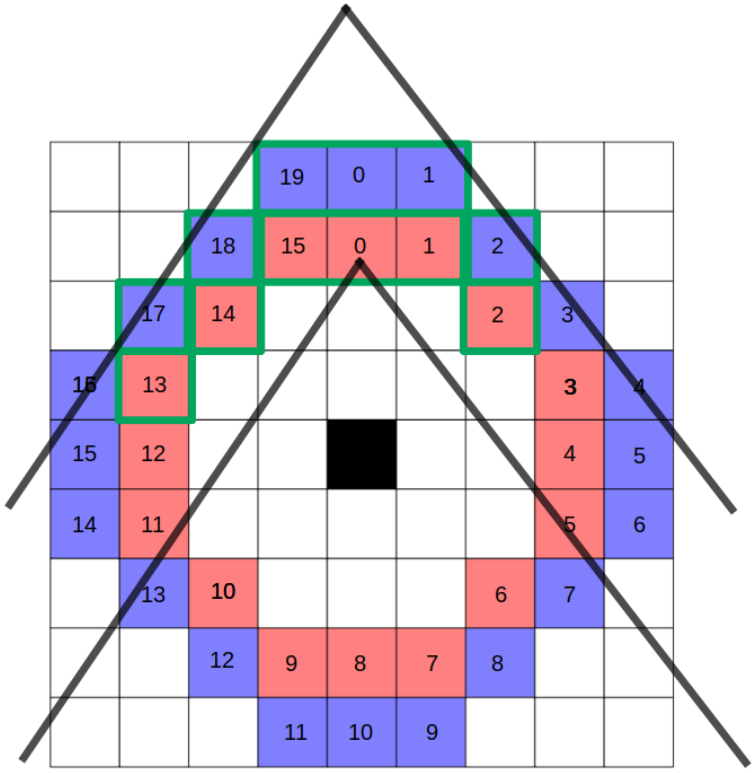



**Figura 6.18** Cadenas que produce una esquina con orientación vertical. Figura adaptada de [29].

# 7 Conclusión y trabajos futuros

Finalmente, comentaremos las conclusiones derivadas de la realización de este proyecto, y hablaremos de algunos posibles trabajos que lo pueden continuar.

## 7.1 Conclusión

En este trabajo, hemos desarrollado un entorno innovador para el procesamiento de eventos que, hasta donde sabemos, no ha sido implementado previamente de esta forma. Para lograrlo, hemos utilizado la arquitectura de computación *Zynq Ultrascale+*, conocida por su capacidad de procesamiento heterogéneo de datos. Esta plataforma nos ha permitido diseñar un flujo de comunicación eficiente que integra de manera óptima dos sistemas distintos: los microprocesadores y las FPGA. El enfoque que adoptamos no solo mejora significativamente la eficiencia temporal y energética del procesamiento, sino que también optimiza el rendimiento general del sistema, aprovechando al máximo las características únicas de cada componente.

La comunicación entre los microprocesadores y las FPGA se realiza mediante DMA, un método que permite la transferencia de datos de manera rápida y eficiente sin involucrar directamente al procesador, lo que reduce la carga de trabajo del sistema y mejora la velocidad de procesamiento. Para implementar esta comunicación, fue necesario desarrollar un módulo específico integrado en el kernel del sistema operativo, generado con *Petalinux*. Este módulo interactúa con una aplicación de usuario que, además, utiliza un controlador (*driver*) especializado para la cámara de eventos. En caso de no disponer de la cámara real, la aplicación puede funcionar como un emulador, lo que facilita la validación del sistema en diferentes escenarios.

Una vez que los datos son procesados, se transfieren a un ordenador externo, donde se pueden visualizar los resultados. Estos resultados, obtenidos y analizados experimentalmente en este trabajo, demuestran las capacidades y el rendimiento del entorno de procesamiento que hemos desarrollado.

Las cámaras de eventos, por otro lado, son sensores emergentes que están ganando popularidad en el campo de la robótica debido a su capacidad para capturar información a gran velocidad y con alta precisión. Sin embargo, el gran ancho de banda que generan estos sensores suele ser un desafío para los sistemas de procesamiento convencionales, que no siempre pueden manejar la gran cantidad de datos en tiempo real. Nuestro trabajo establece una base sólida para el desarrollo de algoritmos más complejos que puedan aprovechar el alto grado de paralelismo que ofrece esta tecnología, permitiendo un procesamiento más rápido y eficiente.

A pesar de haber alcanzado los objetivos planteados, este proyecto ha requerido un considerable consumo de tiempo y esfuerzo. El carácter novedoso tanto de la plataforma *Zynq Ultrascale+* como de la cámara de eventos hace que la documentación y los recursos disponibles sean limitados, lo cual representó un desafío significativo durante el desarrollo. Sin embargo, a pesar de estas dificultades, el aprendizaje adquirido ha sido inmenso. La experiencia obtenida en el desarrollo de este proyecto ha ampliado enormemente mi conocimiento y habilidades en el campo del procesamiento de datos y sistemas embebidos, lo cual será sin duda una ventaja considerable en mi futuro profesional.

## 7.2 Trabajos futuros

La elaboración de este trabajo puede ser el punto de partida para futuras investigaciones y desarrollos que aprovechen sus contribuciones. Al establecer un marco de trabajo sólido y bien definido, se abre la posibilidad de explorar nuevas aplicaciones y mejoras en el procesamiento de eventos en diversos contextos.

Con el *framework* que hemos creado, la implementación de nuevos algoritmos de procesamiento de eventos será mucho más accesible y eficiente. Gracias a la estructura modular y flexible del entorno, los investigadores y desarrolladores solo necesitarán programar nuevos módulos en lenguaje de descripción de *hardware* para integrar algoritmos adicionales. Esto significa que no será necesario modificar las demás capas del sistema, lo que facilita la incorporación de nuevas funcionalidades y reduce significativamente el tiempo de desarrollo y prueba. Se pueden incorporar etapas de procesamiento posterior al detector de esquinas implementado en este trabajo, teniendo en consideración la limitación de recursos de la placa de desarrollo. Sin embargo, si estos fueran insuficientes, el sistema puede migrarse fácilmente a otra placa con mayor capacidad.

Además, el entorno que hemos desarrollado es altamente adaptable, lo que permite modificarlo fácilmente para manejar diferentes flujos de procesamiento. Esto flexibilidad no sólo es útil para la integración de distintos algoritmos de procesamiento de eventos, sino que también permite trabajar con una variedad de cámaras y sensores diferentes. Por ejemplo, el sistema puede ser ajustado para funcionar con cámaras de diferente tecnología o con sensores que capturen datos distintos a los eventos visuales, como sensores infrarrojos o de ultrasonido. Esta adaptabilidad abre un amplio espectro de posibilidades para la aplicación del sistema en diferentes campos, como la automoción, la seguridad, la salud, y la robótica industrial, entre otros.

En resumen, el trabajo presentado no solo ofrece una solución innovadora para el procesamiento de eventos, sino que también sienta las bases para una plataforma expansible y adaptable. Esta plataforma puede evolucionar a medida que se desarrollen nuevos algoritmos y se introduzcan nuevas tecnologías de sensores.

# Índice de Figuras

# Bibliografía


[1] Amara Amara, François Amiel, and Temam Ea, *FPGA vs. ASIC for low power applications*, Microelectronics Journal **37** (2006), no. 8, 669–677.

[2] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi, *Performance comparison of fpga, gpu and cpu in image processing*, 2009 International Conference on Field Programmable Logic and Applications, 2009, pp. 126–131.

[3] M. T. Aung, R. Teo, and G. Orchard, *Event-based plane-fitting optical flow for dynamic vision sensors in fpga*, Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (Florence, Italy), IEEE, 2018, pp. 1–5.

[4] Robert Berner, Christian Brandli, Michael Yang, Shih-Chii Liu, and Tobi Delbruck, *A 240×180 10 mw 12 us latency sparse-output vision sensor for mobile applications*, Proceedings of the 2013 Symposium on VLSI Circuits (Kyoto, Japan), IEEE, June 2013, 12–14 June 2013, pp. C186–C187.

[5] A. Bisulco, F. Cladera, V. Isler, and D. D. Lee, *Near-chip dynamic vision filtering for low-bandwidth pedestrian detection*, Proceedings of the 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (Lyon, France), IEEE, 2020, pp. 234–239.

[6] Dario Cazzato and Flavio Bono, *An application-driven survey on event-based neuromorphic computer vision*, Information **15** (2024), no. 8.

[7] Bharatesh Chakravarthi, Aayush Atul Verma, Kostas Daniilidis, Cornelia Fermuller, and Yezhou Yang, *Recent event camera innovations: A survey*, Proceedings of the ECCV 2024 Workshop on Neuromorphic Vision: Advantages and Applications of Event Cameras (NeVi), 2024, Accepted for publication.

[8] Ping Dai, Yifan Zhang, Xiaoyang Yu, Xue Lyu, and Xiaoyang Qi, *Hybrid neural rendering for large-scale scenes with motion blur*, Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), June 2023, pp. 154–164.

[9] M. Joy Daniel and K. Siva M.E Kumar, *Recent trends and improvisations in fpga*, IOSR Journal of Electrical and Electronics Engineering (IOSR-JEEE) **12** (2017), no. 3, 33–38.

[10] S. M. P. Dinakarrao, *Self-aware power management for multi-core microprocessors*, Sustainable Computing: Informatics and Systems **29** (2021), 100480.

[11] F. Eibensteiner, J. Kogler, and J. Scharinger, *A high-performance hardware architecture for a frameless stereo vision algorithm implemented on a fpga platform*, Proceedings of the

2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (Columbus, OH, USA), IEEE, 2014, pp. 637–644.

[12] S. Ellinidou, G. Sharma, J. M. Dricot, and O. Markowitch, *A sdn solution for system-on-chip world*, Proceedings of the 2018 5th International Conference on Software Defined Systems (SDS 2018) (Barcelona, Spain), IEEE, April 2018, pp. 14–19.

[13] Ralph Etienne-Cummings and Jan Van der Spiegel, *Neuromorphic vision sensors*, Sensors and Actuators A: Physical **56** (1996), 19–29.

[14] Daniele Falanga, Klaus Kleber, and Davide Scaramuzza, *Dynamic obstacle avoidance for quadrotors with event cameras*, Science Robotics **5** (2020), no. 40, eaaz9712.

[15] U. Farooq et al., *Tree-based heterogeneous fpga architectures*, FPGA Architectures: A Survey, Springer Science+Business Media, 2012, pp. 25–46.

[16] M. Humenberger, S. Schraml, C. Sulzbachner, A. N. Belbachir, A. Srp, and F. Vajda, *Embedded fall detection with a neural network and bioinspired stereo vision*, Proceedings of the 2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (Providence, RI, USA), IEEE, 2012, pp. 60–67.

[17] Tomasz Kryjak, *Event-based vision on fpgas – a survey*, IEEE Access **12** (2024), 123456–123467, Senior Member IEEE, Embedded Vision Systems Group, AGH University of Krakow, Poland.

[18] Ian Kuon, Russell Tessier, and Jonathan Rose, *Fpga architecture: Survey and challenges*, Foundations and Trends® in Electronic Design Automation **2** (2008), no. 2, 135–253.

[19] Ian Kuon, Russell Tessier, and Jonathan Rose, *Fpga architecture: Survey and challenges*, Foundations and Trends in Electronic Design Automation **2** (2008), no. 2, 135–253.

[20] Philip H. W. Leong, *Recent trends in fpga architectures and applications*, Proceedings of the 4th IEEE International Symposium on Electronic Design, Test & Applications (DELTA), IEEE, 2008, pp. 195–200.

[21] Haibo Li, Hong Yu, Dawei Wu, Xueliang Sun, and Li Pan, *Recent advances in bioinspired vision sensor arrays based on advanced optoelectronic materials*, APL Materials **11** (2023), no. 8, 081101.

[22] You Li, Julien Moreau, and Javier Ibanez-Guzman, *Emergent visual sensors for autonomous vehicles*, IEEE Transactions on Intelligent Transportation Systems **24** (2023), no. 5, 4716–4737.

[23] Peter Lichtsteiner, Christoph Posch, and Tobi Delbruck, *A 128 x 128 120 db 30 mw asynchronous vision sensor that responds to relative intensity change*, Proceedings of the 2006 IEEE International Solid-State Circuits Conference - Digest of Technical Papers (San Francisco, CA, USA), IEEE, February 2006, 6–9 February 2006, pp. 2060–2069.

[24] A. Linares-Barranco, F. Gomez-Rodríguez, V. Villanueva, L. Longinotti, and T. Delbruck, *A usb3.0 fpga event-based filtering and tracking framework for dynamic vision sensors*, Proceedings of the 2015 IEEE International Symposium on Circuits and Systems (ISCAS) (Lisbon, Portugal), IEEE, 2015, pp. 2417–2420.

[25] M. Liu and T. Delbruck, *Block-matching optical flow for dynamic vision sensors: Algorithm and fpga implementation*, Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS) (Baltimore, MD, USA), IEEE, 2017, pp. 1–4.

[26] M. Liu, W. Kao, and T. Delbruck, *Live demonstration: A real-time event-based fast corner detection demo based on fpga*, 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (Los Alamitos, CA, USA), IEEE Computer Society, jun 2019, pp. 1678–1679.

[27] Florian Mahlknecht, Daniel Gehrig, James Nash, Franz M. Rockenbauer, Ben Morrell, Julien Delaune, and Davide Scaramuzza, *Exploring event camera-based odometry for planetary robots*, IEEE Robotics and Automation Letters **7** (2022), no. 4, 8651–8658.

[28] D. P. Moeys, T. Delbruck, A. Rios-Navarro, and A. Linares-Barranco, *Retinal ganglion cell software and fpga model implementation for object detection and tracking*, Proceedings of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS) (Montreal, QC, Canada), IEEE, 2016, pp. 1434–1437.

[29] Elias Mueggler, Chiara Bartolozzi, and Davide Scaramuzza, *Fast event-based corner detection*, British Machine Vision Conference (BMVC), 2017.

[30] Christoph Posch, Daniel Matolin, and Robert Wohlgenannt, *A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds*, IEEE Journal of Solid-State Circuits **46** (2010), no. 1, 259–275.

[31] B. Ramesh, A. Ussa, L. D. Vedova, H. Yang, and G. Orchard, *Pcarect: An energy-efficient object detection approach for event cameras*, Computer Vision – ACCV 2018 Workshops (G. Carneiro and S. You, eds.), Springer International Publishing, Cham, 2019, pp. 434–449.

[32] A. Scionti, S. Mazumdar, and A. Portero, *Towards a scalable software defined network-on-chip for next generation cloud*, Sensors **18** (2018), no. 8, 2330.

[33] R. Tapia, J.R. Martínez-de Dios, A. Gómez Eguíluz, and A. Ollero, *ASAP: Adaptive transmission scheme for online processing of event-based algorithms*, Autonomous Robots **46** (2022), 879–892.

[34] W. C. Tsai, S. J. Chen, Y. H. Hu, and M. Lun Chiang, *Network-cognitive traffic control: A fluidity-aware on-chip communication*, Electronics **9** (2020), no. 11, 1667, CrossRef.

[35] Michal Turčaník, *Fpga – disruptive technology for military applications*, Proceedings of the International Conference for Scientific Paper AFASES (Brașov, Romania), AFASES, 2012, Conference Paper.

[36] Valentina Vasco, Arren Glover, and Chiara Bartolozzi, *Fast event-based harris corner detection exploiting the advantages of event-driven cameras*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016.

[37] J. N. Yasin, S. A. S. Mohamed, M. H. Haghbayan, J. Heikkonen, H. Tenhunen, M. M. Yasin, and J. Plosila, *Night vision obstacle detection and avoidance based on bio-inspired vision sensors*, Proceedings of the 2020 IEEE Sensors, 2020, pp. 1–4.

[38] M. Zhai, X. Xiang, N. Lv, and X. Kong, *Optical flow and scene flow estimation: A survey*, Pattern Recognition **114** (2021), 107861.